

①

ADA 129755

DTIC FILE COPY

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

80 83 06 23 120

DTIC
JUN 24 1983
A

FINAL REPORT

RESEARCH ON SECURE SYSTEMS AND AUTOMATIC PROGRAMMING

Period: March 1, 1973 to August 31, 1977

Co-Principal Investigators:

**Saul Amarel
(201) 932-3546/2001**

**C. V. Srinivasan
(201) 932-2019/2001**

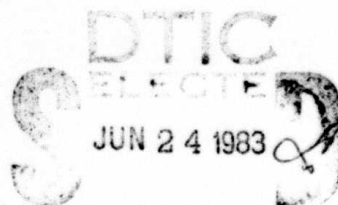
**ARPA Order Number 2406
Program Code Number 3030**

Grant Number DAHC15-73-G-6

**APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED**

**Department of Computer Science
Rutgers, The State University
New Brunswick
New Jersey 08903**

October 14, 1977



INTRODUCTION

The research covered by this final report has two principal objectives. One is the development of methods and tools for the design and evaluation of SECURE SYSTEMS. The other is the development of systems for AUTOMATIC PROGRAMMING with emphasis on programs for specifying, designing, and optimizing programs. In this area we are concerned both with specialized systems for restricted domains and with the development of more general KNOWLEDGE-BASED SYSTEMS.

The various projects in our research program, and the senior investigators in each project are listed below:

A. SECURE SYSTEMS

- (A.1) Protection and Integrity of Data Bases
(Naftaly Minsky)
- (A.2) Architectural Features for Operating System Security
(Manfred Ruschitzka)

B. AUTOMATIC PROGRAMMING

B.1 Studies and Systems in Specific Domains

- (B.1.1) Automatic Optimization of Programs Defined as Finite State Machines
(Edward Wilkens)
- (B.1.2) Programs for Computer Aided Formulation and Improvement of Recursively Defined Algorithms
(Marvin Paul)
- (B.1.3) Computer Aided Design of Specialized Efficient Algorithms for Sorting and Selection Problems
(Saul Levy)
- (B.1.4) Automatic Program Formation from Problem Specifications
(Saul Amarel)

- (B.2) Knowledge Based Systems: The Meta Description System MDS
(Chitoor V. Srinivasan)

I. PROJECT SUMMARIES

A summary status of each project is presented in this section. It is mainly intended as a guide to the collection of technical documents that constitute the body of the present report; it also outlines the main achievements, the current state of work, and the implementation status of computer systems developed in each of the projects.

(A.1) Protection and Integrity of Data Bases

The general objective of this research has been the development of a protection theory which is suitable for the protection of the security and the semantic integrity of databases. This objective has been achieved by means of the new operation-control (OC) protection scheme, described in (Minsky, 1976d, 1976e, 1977a)*. The new scheme is superior to the standard schemes in many ways: it has more expressive power, it is more efficient, more stable and easier to prove and review. In the context of the new scheme we attacked a number of old and new protection issues. In particular: the principle of attenuation of privileges (Minsky, 1977d), and revocation problems (Minsky, 1977a), cooperative authorization (Minsky, 1977b), "secure-information flow" (Minsky, 1976c), and others. We studied the application of our scheme in a number of specific domains, such as: privacy protection and its "intentional resolution" (Minsky, 1976 a), networks, management and auditing of financial systems (Minsky, 1976e and 1977c).

Fifteen publications covering work in this area are included in the present final report.

Most of the work has been theoretical; however, it was supported by two major system development activities:

- a) A system which implements "files with semantics".
(This was written in SIMULA and is now completed.)
- b) An experimental database system based on the OC scheme.
(This system has not yet been finished but parts of it are working.)

(A.2) Architectural Features for Operating System Security

The main thrust of our research effort in the area of Secure Operating Systems resulted in the definition of a novel protection mechanism which attempts to combine the advantages of the capability and access control list schemes and to minimize their limitations. The properties of this mechanism (Ruschitzka, 1977a) have been studied in the context of a

*References are listed in Section II below (List of Publications).

simulation which is based on the design of an educational computer system (Ruschitzka, 1977b). This system is characterized by an integrated design philosophy with respect to the hardware architecture (Ruschitzka, 1977c, 1977d), the implementation language (Ruschitzka, 1977e), and the operating system (Ruschitzka, 1977f).

The protection of information resources depends on their secure management. Our work on Secure Operating Systems has been influenced by our earlier efforts in research management, in particular scheduling, which have continued in parallel.

Eight articles covering work in this project are included in the present final report. Two of these (Ruschitzka, 1977g, 1977h) are in the area of scheduling, and they are included to reflect the overall scope of our activities in this project.

(B.1.1) Automatic Optimization of Programs Defined as Finite State Machines

This research has achieved positive results in two areas, the building of an optimizer for the efficient encoding of finite state machines, and the definition and implementation of a specification language for finite state machines.

The primary thrust of the work on an optimization program, which has been the subject of most of the effort in this area, has been to produce a program which can optimize finite state machines built by practitioners of practical programming, rather than toy state tables designed to exemplify the techniques. Such practical state tables should have a range of inputs times states product in the area of fifty to one thousand. When this product is above one thousand, and preferably even less, principles of good modularization suggest that a composite of several such state tables be used.

Part of the objective of this work was to extend the switching theoretic partition methods to include such practical considerations as don't care values in the state table. Switching theory has not been found applicable in the past to problems of realistic size, without including such difficulties as don't cares. The approach taken was to define constructs that had most of the mathematical properties of partitions, but to sacrifice elegant properties to the more pragmatic goal of achieving solutions to large programs.

An algorithm was developed based around these partition theory techniques that uses a very complex, highly pruned search for an optimal solution. The search is ordered in such a way that when a solution is found it is optimal under the criteria used. This algorithm is described partially in (Wilkins, 1977a). This paper describes the algorithm as it was when it was first able to find solutions to small state tables. The algorithm as presently implemented is at least ten times faster than the

original algorithm. The speed-up is due to increased pruning of the search tree by some as yet unreported techniques. This final algorithm approaches the goal of solving practical sized problems by completing state tables with sizes of 96 and 120 entries in 7.2 and 16 seconds respectively on an IBM 370/168. However, it did not complete state tables of size 297 or 616 in 2 minutes on this machine. The larger state tables we have been using for tests have been drawn from the applications literature, rather than arbitrarily constructed state tables. This accounts for the lack of a large number of samples on which to evaluate the overall performance of the algorithm. The search traces of the two incomplete state tables suggests still further pruning techniques that might be applied.

In summation, the results of this work suggest that realistic sized state tables can be optimized. Further, the techniques used may be more broadly applied to different criteria of optimality. In addition, by the level of success in finding a solution to optimization, it is reasonable to conclude that sub-optimal, good solutions can be found with less development effort and less computer time.

The Finite State Specification Language developed as part of this work provides a powerful yet simple to learn language. Since it is based around the concept of a finite state machine, it is of potential value to the hardware designers who are increasingly forced to learn micro-processor assembly language. The use of this language removes most of the programming from the task, since only a state table is necessary. The language is described in (Wilkins, 1977b), and more fully, and with more description of its incorporation into a system in (Wilkins, 1977c).

The above described programs have been coupled with a small program that runs on a PDP-10 to combine the outputs of the language and optimizer and produces assembly code with the encoded finite state machine and its interpreter ready to be assembled and run on a PDP 11. The machine dependent (PDP-11) part is about a page of fortran code. Therefore this is easily adapted to other target computers.

Three papers, covering work in this area, are included in the present final report.

(B.1.2) Programs for Computer-Aided Formulation and Improvement of Recursively Defined Algorithms

Our objective has been to develop a system which will aid a user in the specification of a recursive definition for an enumeration-based algorithm, and then to provide the user with one or more 'good' algorithms that implement the definition and/or suggestions for reformulating the initial definition. Our research in this area produced several theoretical results which provided the basis for building a small experimental system which transforms a given recursive definition into efficient code.

In (Paull, 1977a, 1977b) an attempt is made to formally and succinctly state and prove a number of relations between recursive definitions and their implementations, including the significance of the 'uniform inverse' in allowing storage savings, and of transformations which convert a recursive definition into a set of equations which can be solved speedily.

In (Paull, 1977c, 1977d) we present a generalization of the principle which allows the time efficient implementation of the shortest path problem (Dijkstra's algorithm). This principle is applied to two problems involving solutions of sets of equations and it yields algorithms more efficient than those currently known.

The experimental system which we constructed consists of the following programs:

OFN.SNO. A SNOBOL program (~ 3300 statements) which takes as input a recursive definition; it searches for 'uniform inverse' and other properties, and it produces an output program to implement the given definition.

SIM.SNO. A SNOBOL program (~ 1000 statements) which simulates the recursive definition for sample inputs and produces the resultant output.

CHART.SNO. A SNOBOL program (~ 500 statements) which makes flowchart representations of output of OFN.SNO.

INF.SNO. A SNOBOL program (~ 200 statements) which contains descriptions and instructions on use of OFN.SNO.

Four papers covering work in this project are included in the present final report.

(B.1.3) Computer Aided Design of Specialized Efficient Algorithms for Sorting and Selection Problems

This research was directed to the automatic generation of algorithms for solving a class of problems involving the order of a given set of numbers. The progress in this report is described in (Levy, 1977).

We produced two ILISP programs, NAA and NAA1, which generate good and extremely good programs (respectively) for determining order statistics. In response to the request "Find the I-th out of R numbers" NAA produces all 'reasonable' candidate programs and selects the best among them. NAA1 is a refinement on NAA which not only produces the best programs generated by NAA but also uses certain heuristics which enables it to produce more human-like programs; in fact for all the examples which we have tried NAA1 had produced the best programs known for the solution of those problems.

We started working on two additional aspects of the problem: producing a better interface to the NAA programs (more intelligible output), and an approach to the generation of programs for the solution of order statistics problems in the presence of additional information about the input set (e.g., find the I-th largest of (a, b, c, d, e, ..., h) given that it is not c, d, or h). We are concerned with the assimilation of the information into the program generating process and consider this a key problem in the design of a man-machine interactive program-generating system. Work in this area did not reach yet the state of documentation.

One report covering work in this project is included in the present final report.

(B.1.4) Automatic Program Formation from Problem Specification

The long term objective of this research is to understand the interactions between knowledge representations and reasoning strategies in program synthesis problems. In our work (which is also supported by NIH as part of the Rutgers Research Resource on Computers in Biomedicine) we considered several forms of specification for the program synthesis problem; but the emphasis has been on specifications in the form of a finite number of input/output correspondences.

We developed a strategy which combines a model-guided global approach to search for a program with a local approach which focuses on detailed modifications of a program based on analysis of its performance. To test this strategy, we started to implement a computer system (in LISP 1.6) which we called TF. Our main effort in this project over the last 1½ years was directed to representations in TF and to subsystems for acquiring knowledge and for building and managing the knowledge base of TF. The growth in size and complexity of TF (which occupies now a core image of about 120K words in a PDP-10) induced us to direct attention to several problems of large system development. Because of the relatively small level of effort directed to this project and the unexpected requirements of the system building task, we did not reach as yet a stage where our program formation strategies could be studied experimentally. We plan to continue work in this area, and expect to produce reports documenting the TF system and related studies in program formation within about a year.

(B.2) Knowledge Based Systems; The Meta Description System MDS

MDS proposes a new way of organizing intelligent systems and a new approach to Automatic Programming--where knowledge in a domain is used to construct programs from vague specifications of their requirements. We call this "Knowledge Based Programming" (Srinivasan, 1973b). In constructing programs in a domain and in using the domain knowledge, MDS can learn from its experience in a domain and improve its performance. Establishing the logical foundations of these processes for knowledge

handling and utilization has been the major contribution of our work in this project. Details of these processes are discussed in (Srinivasan, 1977). This is the principal report which summarizes the progress, the key contributions and the significance of work in MDS. Problems are classified in MDS as belonging to one or more of the following categories: Assimilation, Planning, Recognition, Goal Seeking, Theorem Proving, and Language Understanding.

Our recent work on Theorem Proving (Sandford, 1977a, 1977b) establishes the logical basis for using model based reasoning to guide theorem proving search. It also presents a new, sound and complete resolution refinement strategy, called Hereditary Lock Resolution.

The MDS formalism for describing domain knowledge has been applied to several tasks. Applications to banking and medicine are described in (Srinivasan, 1974) and (Irwin and Srinivasan, 1976). A system which is a specialization of MDS for research in action interpretation problems was developed in the framework of our NIH-supported Research Resource on Computers in Biomedicine and it is being applied to work in modeling Belief System (the system is called AIMDS). For the period 1973 to 1976, research on MDS was jointly supported by the ARPA grant and by the NIH grant.

Considerable progress was made towards the implementation of MDS. Major parts of the system have been written in INTERLISP. System development proceeded at ISI and at SUMEX-AIM through the ARPANET. At present the system is at SUMEX-AIM. The current status of the implementation may be summarized as follows: The "Domain Definition" system is now ready. This is the subsystem that enables a user to define "Structural", "Sense", and "Transformational" knowledge in a domain and have the system create compact representations of these definitions in the model space for the defined domain. The second subsystem that is now operational is the "Dimensional Consistency Check" system. This checks the given domain definitions for "Structural Consistency" and builds a dictionary of all possible interactions that can occur in the model space of the domain, between properties of various instantiated models, and for each such interaction between pairs of properties of objects, builds the conditions under which the interaction would occur. To complete the model space management system it is still necessary to complete the so called "Domain Compiler". Based on the domain definitions and the interactions between the various objects in the domain and the conditions on the interactions, the domain compiler compiles LISP code for the instantiation of the various objects and relations in the model space, and for the access and updating of the objects and relations. The compiled code has two parts: one for the creation and updating of models in the model space; and the other for the checking of consistency of the model space itself. We have now devised a scheme for compiling the consistency conditions. Most of the code for the model space compiler has now been written. The various parts of this code should now be debugged and properly integrated. We expect to complete the remaining tasks of the implementation in 1978 (which is beyond the period of the current grant).

Twelve papers covering work in this area are included in the present final report.

October 14, 1977

APPENDIX

Financial Summary

Research on Secure Systems and Automatic Programming

I. Financial Status

A. Total funds allocated for the period March 1, 1973 to August 30, 1977		\$554,185.
B. Expenditures during period March 1, 1973 to June 29, 1974	\$ 92,226.	
Expenditures during period June 30, 1974 to August 31, 1977	<u>461,875.</u>	
Total expenditures for the period March 1, 1973 to August 31, 1977	\$554,101.	<u>54,101.</u>
Difference		84.

II. Summary breakdown of the expenditures for the last reporting
period of the grant (June 30, 1977 to August 31, 1977)

	Total Expenditures 7/1/74 to 6/29/77	Expenditures This Period 6/30/77 to 8/31/77	Total Expenditures 7/1/74 to 8/31/77
PERSONNEL:			
<u>Faculty</u>			
Levy, S.	\$ 12,221	\$ 2,450	\$ 14,671
Minsky, N.	10,559	4,228	14,787
Morgenstern, M.	-	1,770	1,770
Paul, M.	18,070	-	18,070
Ruschitzka, M.	6,810	3,910	10,720
Srinivasan, C. V.	8,856	-	8,856
Wilkins, E.	<u>11,742</u>	<u>-</u>	<u>11,742</u>
	\$ 68,258	\$ 12,358	\$ 80,616
<u>Adjuncts</u>			
Morrell, L.	1,616	-	1,616
Groeber, B.	1,186	-	1,186
Anand, P.	<u>4,373</u>	<u>-</u>	<u>4,373</u>
	\$ 7,175	-	\$ 7,175

October 14, 1977

	Total Expenditures 7/1/74 to 6/29/77	Expenditures This Period 6/30/77 to 8/31/77	Total Expenditures 7/1/74 to 8/31/77
<u>Research Assistants</u>			
Brutman, N.	\$ 5,619	\$ -	\$ 5,619
Chen, D.	7,670	1,852	9,522
Dowuona, N.	17,440	1,940	19,380
Goegelman, M.	-	1,760	1,760
Griffin, G.	6,846	-	6,846
Grissom, J.	11,640	-	11,640
Sakrowitz, M.	1,386	-	1,386
Sandford, D.	2,284	-	2,284
Yuan, J.	488	-	488
	<u>\$ 53,373</u>	<u>\$ 5,552</u>	<u>\$ 58,925</u>
<u>Clerical Support</u>			
Wiese, B.	\$ 1,662	\$ 1,260	\$ 2,922
<u>Terminated Personnel</u>			
	<u>126,953</u>	<u>-</u>	<u>126,953</u>
<u>Salary Totals</u>			
	<u>\$257,421</u>	<u>\$ 19,170</u>	<u>\$276,591</u>
<u>Fringe Benefits</u>			
	<u>28,138</u>	<u>227</u>	<u>28,365</u>
PERSONNEL TOTAL	<u>\$285,559</u>	<u>\$ 19,397</u>	<u>\$304,956</u>
<u>EQUIPMENT</u>			
1 Special IMP-Host Interface	\$ 12,500	\$ -	\$ 12,500
Computer Services	60,000	1,000	61,000
Computer Supplies	7,160	356	7,516
Data Communications	<u>7,883</u>	<u>443</u>	<u>8,326</u>
EQUIPMENT TOTAL	<u>\$ 87,543</u>	<u>\$ 1,799</u>	<u>\$ 89,342</u>
<u>OTHER EXPENSES</u>			
Publication Costs	\$ 4,224	\$ 415	\$ 4,639
Travel	<u>7,254</u>	<u>366</u>	<u>7,620</u>
OTHER EXPENSES TOTAL	<u>\$ 11,478</u>	<u>\$ 781</u>	<u>\$ 12,259</u>
TOTAL DIRECT COSTS	<u>\$384,580</u>	<u>\$ 21,977</u>	<u>\$406,557</u>
Overhead (20% Salaries)	<u>\$ 51,484</u>	<u>\$ 3,834</u>	<u>\$ 55,318</u>
TOTAL ALL EXPENDITURES	<u>\$463,064</u>	<u>\$ 25,811</u>	<u>\$461,875</u>
Expenditures during period March 1, 1973 to June 29, 1974			<u>92,226</u>
			<u>\$554,101</u>

October 14, 1977

Notes on Financial Status

1. In our personnel plan, we had budgeted fractions of faculty time to work on the project during the academic year. The faculty monies thus released have been used to compensate adjuncts who relieve the faculty on the project from some of their teaching responsibilities. This arrangement permitted faculty to spend the planned time on research.
2. This is the final reporting period for the grant. This grant has covered the period March 1, 1973 to December 29, 1976, with two no-cost extensions, the first to June 29, 1977 and the second extension to August 31, 1977. The initial award totalled \$557,670, of which \$3,485 was not funded. Thus the actual available grant was \$554,185. Our expenditures to date from the beginning of the grant (March 1, 1973) are estimated to be \$554,101. The University has continued to cover the difference between actual computing costs and grant funds, as in the past.

* * *

II. LIST OF PUBLICATIONS*

[Note: For convenience, the publications are listed alphabetically by author; they are also indexed in the left margin by project. For example, (A.1) indexes an entry from the project on "Protection and Integrity of Data Bases".]

(B.1.3) Levy, S. (1977) "Approaches to Automatic Program Generation," Report SOSAP No. TR-36, Department of Computer Science, Rutgers University, July 1977.

(A.1) Minsky, N., (1973) "On the Security of Data Base Systems". SOSAP No. TR-1, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, December 1973.

Minsky, N., (1974a) "Comments on Privacy of Data Bases". SOSAP No. TR-8, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, April 1974.

Minsky, N., (1974b) "On the Formation of Abstract Data Types". SOSAP No. TR-10, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, July 1974.

Minsky, N., (1974c) "Protection of Data-Bases, and the Process of User Data-Base Interaction". SOSAP No. TR-11, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, September, 1974.

Minsky, N., (1974d) "Another Look at Data-Bases," Bulletin of SIGMOD, September 1974.

Minsky, N., (1974e) "On Interaction with Data Bases," proc. of the ACM SIGMOD Conference, May 1974.

Minsky, N., (1976a) "Intentional Resolution of Privacy Protection in Database Systems," Management/Database Systems, Vol. 19, Number 3, pp. 148-159, March 1976.

Minsky, N., (1976b) "Files with Semantics," SOSAP No. TR-17, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, March 1976.

Minsky, N., (1976c) "A Semi-Lattice Model for Secure Information Flow," SOSAP No. TR-24, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, May 1976.

Minsky, N., (1976d) "An Activator-Based Protection Scheme," SOSAP No. TR-25, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, July 1976.

* All the 42 publications in this list are included as part of the present final report, with the exception of three (3) publications (Srinivasan 1974; Srinivasan 1976d; and Srinivasan 1977) which were mailed directly by Dr. Srinivasan (who is currently in India on a sabbatical) to Mr. W. Carlson of IPTO.

Minsky, N., (1976e) "Protection in Programming Languages by Operation-Control," SOSAP No. TR-27, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, September 1976.

Minsky, N., (1977a) "An Operation-Control Scheme for Authorization in Computer Systems," SOSAP No. TR-33, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, April 1977.

Minsky, N., (1977b) "Cooperative Authorization in Computer Systems," SOSAP No. TR-34, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, November 1977.

Minsky, N., (1977c) "Auditing of Computerized Financial Systems," SOSAP No. TR-35, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, July 1977.

Minsky, N., (1977d) "The Principle of Attenuation of Privileges and its Ramifications". To be presented at the workshop on "Foundation of Secure Computers," October 1977. September 1977.

(B.1.2) Paul, M.C., (1977a) "Relations Between Recursive Definitions and Their Efficient Algorithms". SOSAP No. TR-37, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, January, 1977 (submitted for publication).

Paul, M.C., (1977b) "Memory Efficient Implementations of Recursive Definitions". SOSAP No. TR-38, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, May 1977. (Submitted for publication).

Paul, M.C., (1977c) "A Principle Useful in the Design of Minimum Path and Other Algorithms". SOSAP No. TR-39, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, June 1977. (Submitted for publication).

Paul, M.C., (1977d) "The Min-Max Branch in a Graph--An Application of the Minimum Constant Principle". SOSAP No. TR-40, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, July 1977.

(A.2) Ruschitzka, M., (1977a) "On the Use of Licenses as a Protection Mechanism". SOSAP No. TR-35, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, August 1977.

Ruschitzka, M., (1977b) "An Operating Systems Implementation Project for an Undergraduate Course". Proc. Seventh Tech. Symp. on Computer Science Education, Atlanta, Georgia, February 1977. SIGCSE Bulletin 9, 1, pp. 77-84.

Ruschitzka, M., (1977c) COS-Model 1 Reference Manual. CS 416/18 Class Notes, Dept. of Computer Science, Rutgers University, New Brunswick, New Jersey, 1977.

Ruschitzka, M., (1977d) COS-Model 2 Reference Manual. CS 416/18 Class Notes, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, 1977.

Ruschitzka, M., (1977e) CAL-CAROL Reference Manual. CS 416/18 Class Notes, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, 1977.

Ruschitzka, M., (1977f) COSMOS Reference Manual. CS 416/18 Class Notes, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, 1977.

Ruschitzka, M., and R.S. Fabry, (1977g) A Unifying Approach to Scheduling. Comm. ACM 20, 7 (July 1977), 469-477.

Ruschitzka, M., (1977h) "An Analytical Treatment of Policy Function Schedulers". To appear in the special issue "On Interfaces with Computer Science" of Opns. Res. in late 1977 or early 1978.

(B.2) Srinivasan, C.V., (1973a) "Architecture of Coherent Information Systems: A General Problem Solving System". IEEE Vol. C-25, No. 4 pp. 390-402. Also presented at IJCAI 3 in August 1973.

Srinivasan, C.V., (1973b) "Programming Over A Knowledge Base: The Basis for Automatic Programming". SOSAP No. TM-4, Department of Computer Science, Rutgers University, New Brunswick, New Jersey December 1973.

*Srinivasan, C.V., (1974) "Description in MDS of a Coherent Information System for a Banking Domain," SOSAP No. TM-4A, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, June 1974.

Hsu, Tau, (1976) "The Blind Hand Problem". SOSAP No. TM-10, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, December 1976.

Srinivasan, C.V., (1976a) "Introduction to the Meta Description System". SOSAP No. TR-18, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, January 1976.

Srinivasan, C.V., (1976b) "Theorem Proving in the Meta Description System". SOSAP No. TR-20, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, January 1976.

Irwin, J., and C.V. Srinivasan, (1976c) "Description of CASNET in MDS". RUCBM-TR-49, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, August 1976.

*Srinivasan, C.V., (1976c) "Formal Definition of the Model Space of the Meta Description System," SOSAP No. TR-20B, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, October 1976.

*Srinivasan, C.V., (1977) "The Meta Description System: A System to Generate Intelligent Information Systems, PART I: The Model Space". SOSAP No. TR-20C, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, July 1977.

Sandford, D.M., (1977a) "Hereditary-Lock Resolution: A Resolution Revinement Combining a Strong Model Strategy with Lock Resolution". SOSAP No. TR-30, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, April 1977.

Sandford, D.M., (1977b) "Formal Specifications of Models for Semantic Theorem Proving Strategies". SOSAP No. TR-32, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, June 1977.

(B.1.1) Wilkens, E.J., (1977a) "Realizations of Sequential Machines Using Random Access Memory". IEEE Trans. on Computers (to be published).

Wilkens, E.J., (1977b) "Finite State Techniques for Software Engineering Systems - Applications to Microcomputer and Large Scale Systems". Tenth Hawaii International Conference on System Sciences, 1977.

Wilkens, E.J., (1977c) "Finite State Techniques in Software Engineering". IEEE COMPSAC 77, 1977.

* The "starred" publications by C.V. Srinivasan are being mailed directly from India (where Dr. Srinivasan is now on sabbatical) to Mr. W. Carlson of IPTO.

SOSAP-TR-1

December 12, 1973

ON THE SECURITY OF DATA BASE SYSTEMS

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

ON THE SECURITY OF DATA BASE SYSTEMS

N. Minsky

Abstract: Two aspects of security in data bases are discussed in this paper: the description of the *security state*, and its *enforcement mechanism*. The conventional approach to these two aspects has been found inadequate, and a new approach has been proposed. According to the proposed approach, the user's program which interacts with the data base is controlled by a set of rules, collectively called *schema*, whose function is to authorize the operations performed by the user. The schema has no authority over information which is not related to the data base, but it can restrict the way in which information from the data base is manipulated even when such information is in the user's working space.

Acknowledgement: I have benefited greatly from many stimulating discussions with Dr. C. V. Srinivasan, and from the interest and encouragement of other participants of the "security" project in Rutgers.

ON THE SECURITY OF DATA BASE SYSTEMS

N. Minsky

1. Introduction

It is customary to distinguish between two aspects of security of data bases, their *integrity* and *privacy*. Although this classification will not be essential to this paper, we will use it in order to introduce the subject.

In order to maintain the integrity of any system, one must first have a definition of what the proper state of this system is. This obvious observation raises the question of how data bases are defined. Being a dynamic system, it is clear that a data base cannot be defined just by specifying its information content at a certain moment in time; there must also be a set of rules which specify the behavior of the data base under interaction with the outside world. For example, it is not enough to know that there is an entity called "number-of-employees" in a data base; it must be formally specified that this entity should always be equal to (say) the number of "employee" records in the data base.

Unfortunately, this is not the approach employed by contemporary data-base systems. In most of these systems one cannot even specify the scope of an entity, not to mention the ability to formulate general rules of behavior. The dynamic behavior of the data base under these systems is defined, de facto, by the users' programs which interact with it.

There is obviously no hope for providing integrity for data bases as long as this approach prevails. The behavior of the data base should not be controlled by users' programs. On the contrary, the interaction of users' programs with a data base should be under the jurisdiction of a set of rules built into the data base itself.

Privacy has to do with the right of a user to perform certain operations on the data base. (The right to receive information from the data base is the most publicized aspect of privacy, but it is not the only one.) This implies, again, that there is a set of rules in the data base which dictate the ways in which users can interact with it.

Thus, *privacy* and *integrity*--as different as their objectives may be--have one thing in common: they both impose restrictions on the way in which users interact with the data base. The form of these restrictions, and their enforcement, are the subject of this paper.

The main thesis of this paper would be that privacy and integrity of data bases cannot be secured very well just by imposing restrictions on the information which flows into the data base and out. It will be argued that some control must be exerted on the users' programs which manipulate this information. A model for user data-base interaction, which allows for such a control, will be proposed. It should be pointed out at the outset that the objective of this preliminary paper is not to present final results, but rather to point out to difficulties in the conventional approach to security, and to suggest an alternative approach.

2. A Critical Review of Conventional Approaches

When discussing security, it is sometimes useful to distinguish between the following two aspects of it: (a) The specification of the *security state*, by which we mean: the method used to specify the security measures to be imposed. (b) The enforcement of the security state. We will try to evaluate some conventional approaches to security in terms of these two aspects.

2.1 On the Specification of Security States

The prevailing approach to the specification of the security state of systems is essentially the following: Assuming that there is a finite number of objects in the system, and that there is a finite set of potential users (or classes of them), all we have to do is to specify which *actions* can every one of the users apply to every object. One way to represent that is by means of a *matrix* [Con.72, Grh.72, Lam.71]. The element a_{ij} of such an "access matrix" is the set of actions (typically called "accesses") which user i can apply to object j . Although this *access-matrix* model is usually considered to be conceptually complete,⁽¹⁾ it is obviously not always practical. A mathematically equivalent model, which is often more convenient, is the *environment model* ⁽²⁾ which had been recently elaborated by Anita Jones [Jon.73]. According to this model, every user who interacts with the system operates within certain *environment* E which is defined by a set of pairs $\{(a,p)\}$, where P is

(1) In its complete form, the matrix elements a_{ij} in this model include the condition under which user i has this access to object j . But our criticism of the model is independent of such a condition.

(2) Also called the "capability model."

an object in the system and a is the action (access) which the user is allowed to apply to p . (The pair (a,p) is called *access-right*.)

Although this approach may be adequate for operating system; it is not general enough. To see that, consider an operation A which a given user wants to perform. Using Jones' *environment model*, A would be a legal operation if one of the following conditions is satisfied.

a) A is a *legal primitive operation* by which we mean that A calls for the application of an action a to an object p , such that the access-right (a,p) exists in the environment.

b) A is reducible (equivalent) to a sequence of legal primitive operations, to be called the *components* of A .

The difficulty here is that one may want to allow a user to perform a composite operation without authorizing him to perform any of its components. An example may clarify this point.

Let b, b', c, c' be objects in a data base, and let U be a user. Suppose that U is allowed to read these four objects. In addition, he is allowed to exchange b with b' , and c with c' , but he cannot change these objects in any other way. Suppose that there is a procedure EXCH in the data base which exchanges any pair of objects specified as its arguments. U should be allowed to call EXCH(b,b') and EXCH(c,c'), but these operations are not reducible to legal primitive components since U is not allowed to write into the objects b, b', c, c' . Therefore, these operations must be directly specified as legal operations for U . But since they are operations on pairs of objects, there is no way to specify them by means of Jones' environment-model. The most one can do in this model is to allow U to apply the procedure EXCH to any of the objects b, b', c, c' . But this would allow U too much, for example, to exchange b with c .

In other words, the problem with the *matrix* and the *environment* models is that one does not just access objects as is assumed by these models; one performs operations on the data base. Such an operation can be formally described as a procedure q_0 , being applied to a set of arguments $q_1 \dots q_n$; where the procedure, as well as its arguments, may be objects of the data base. Therefore, it is not a matter of the user having a particular access to an object, but of him being able to perform an operation in which several objects of the data-base participate. To specify that such an operation is permitted, we should use something like the tuple $(q_0, q_1 \dots q_n)$, rather than Jones' pair (a,p) .

In the case of operating systems, one does not usually deal with *composite operations*, therefore, the "access-matrix" model may be satisfactory. In the case of *data-bases*, on the other hand, composite operations become very important so that a more general technique for the representation of security states is needed.

2.2 On the Enforcement of Security States

In most papers on security of data bases it is assumed, sometimes implicitly, that data bases can be protected by means of *access-control* [Bro.71, Owe.71]. Indeed the term "access-control" is frequently used as a synonym for "protection." By "access-control" one usually means a mechanism which controls the flow of information into the data base and out. This mechanism is not supposed to have any control over the user's program, to be denoted by π , which manipulates this information and generates the various storage and retrieval requests. This localization of the protection of the protection mechanism is very convenient. Unfortunately, the access-control mechanism does not always work well, as we will see now by an example.

Consider a data base in which there is a set A of objects, and a function F defined over the members of A . Consider a user U whose interaction with the data base has to be restricted by the following two rules:

Rule 1: *Only a subset $A' \subset A$ is accessible to U .*

Rule 2: *U is allowed to apply F only to members a' of A' .*

Rule 1 can be easily enforced; by means of access control one can simply filter out every object which does not belong to A' . Rule 2 may seem to be automatically satisfied; since, by rule 1, U does not have an access to objects of A which do not belong to A' . Unfortunately, there is nothing to prevent the user's program π from getting a copy of a member of A from an outside source, or from making up one by itself.

Thus, rule 2 is not automatically satisfied and must be enforced explicitly. One may try to do this by checking the argument of F for membership in A' every time that F is invoked. This may be involved with a lot of computation and data base search which is, in a sense, a duplication of effort. Moreover, one might not be able to perform such checking at all. To see that, let us change our example slightly. Instead of rule 2 consider:

Rule 2': *π is allowed to apply f only to elements retrieved by π from A' .*

Now, given an invocation $F(a')$ of F , it is not enough to check if $a' \in A'$, because of the following reasons: First, it may be that $a' \in A'$, but the user U did not receive it legally from the data base. Secondly, it may be that when $F(a')$ is invoked, a' is not a member of A' , but it was a member of A' when U got it from the data base. Finally, the set A' may simply be defined as the set of all objects which U retrieved from A , so that there is no "objective" criterion by which one can determine if $a' \in A'$. One can get

around these difficulties by maintaining a record of all members of A' which have been retrieved by π , however, this may be extremely wasteful and can be considered as an entirely unrealistic solution.

It should be pointed out that rules such as 1 and 2 (or 2') appear frequently in practice. Suppose, for example, that A is a set of names of patients in a medical data base. Let the user U be a doctor, and let $A' \subset A$ be the set of names of the patients of U . Let $F(a)$ be a function which returns some personal information about the patient a . In this case rules 1 and 2 simply mean that the doctor is allowed to get personal information about his own patients only, which is a very reasonable restriction. As an example of rule 2', consider the same doctor who wants to transfer a patient of his own to another doctor, to be done by the function F . He does it as follows: A patient name $a' \in A'$ is retrieved, and deleted from A' . When $F(a')$ is invoked, assigning a' to another doctor, it is already too late to verify that a' is a member of A' ; it is not any more.

Thus, it was shown that the enforcement of some common restrictions by means of access-control can only be achieved at the price of gross inefficiency. But the example above also suggests a solution to this problem. If one can guarantee that only objects retrieved from A' can be used by π as parameters of F , then rule 2 (or 2') is automatically satisfied. This, however, requires that π should be restricted in certain ways. The form of such a restriction will be discussed next.

3. A Basic Model for the Interaction of Users with the Data Base.

It has been shown in the last section that access-control alone is not a satisfactory tool for protection of data base. It has also been suggested that one might get better results by exerting some control over users' programs which interact with the data base. In this section we will consider a model for the user data base interaction which admits such a control. (The model to be discussed here is intentionally simplified. Some generalizations of it will be discussed in section 4.)

The main participants in the user data-base interaction are the data-base D , and the user U who is using a program π . This interaction is supervised by a mechanism (or data structure) which we will call schema⁽¹⁾, to be denoted by S . The schema is supposed to contain the *security-state* for a given set of users, namely, the definition of what these users can and cannot do. For a program π , written by a user U , to operate, it must be "connected" to one of the schemas in D . This connection mechanism will not be specified here, but the following is assumed about it: Every schema S is connectable to programs which "belong" to a specific set of users to be called the *patrons* of the schema S , provided that they are written in a given programming language $L(S)$ (or simply, L). (For example, a certain schema in a medical data base may be connectable only to COBOL programs written by doctors.) It is also assumed that once such a connection is done, the correct identity, U , of the user is known to the schema. (This, of course, is a very strong assumption which is very hard to satisfy.)

(1) The concept of *schema* as an interface between user's programs and the data base is a well-known and extremely important part of the data base architecture. (However, it is typically called sub-schema.) It was initially introduced in order to help reducing the dependency of π on the storage structure of the data base [Dbtg.72], but its potential role in maintaining security (usually by access-control) is also well recognized. It is the security role of the schema which would interest us here.

The program π which is connected to the schema S would be denoted by π/S .⁽¹⁾ The way in which π/S is controlled by S is discussed next.

3.1 A Model for Users' Programs

The program π/S operates within an *environment* defined jointly by the schema S and by the programming language L in which π is written. This environment consists essentially of a set Q of *objects* available to π , and a set R of *rules* which specify the operations that π can apply to objects in Q .

The set Q is a union of the following three disjoint sets:

a) The set Q_L of *primitive objects*. These are the primitives of the programming language L . For example, a primitive operator such as "+", or a constant such as "179".

b) The set Q_S of *permanent objects*. These are objects which belong to the the data base, and which are made available to π/S by the schema S . Such objects may be, for example, files and procedures which operate on them. They are "permanent" only in the sense that their existence does not depend on the existence of a program π which interacts with them.

c) The set Q_T of *transient objects*, objects which are generated by π itself. Such an object exists only in the context of the program which generated it, and it disappears with the program.

As to the structure of the objects we will only say this: Every object $q \in Q$ is a pair

$$q = (\text{brand}, \text{value})$$

(1) We will occasionally write π instead of π/S

where the *brand* is a symbol which is a member of a set B , to be introduced below. The role of the brands in our model which will be explained in detail later would be similar, but not identical, to the role of *types* in most programming languages. The rest of the object, which is called its *value*, can be any data structure, e.g., a number, a file or a procedure.

The set of *brands* B is a set of symbols. It is a union of two sets, $B = B_L \vee B_S$, as we will yet see B_L would be part of the specification of the language L , and B_S would be part of the schema S .

The following can be said about the *brands* of the three classes of objects introduced below:

- a) The brands of objects $q \in Q_L$ will be members of B_L .
- b) The brands of the objects of Q_S would be the names of these objects themselves. (This is a rather formalistic point; these brands will not be actually used, and we therefore do not include the names of the objects of Q_S in the set of brands).
- c) The objects in Q_T may have any brand from B , according to the rules which will be introduced later.

Some conventions about brands are in order:

- a) To distinguish between brands and other symbols which will appear in our discussion, we will denote the former by a bar, e.g., \bar{b} (generic brands will be denoted without a bar).
- b) The set B_L would always contain the distinguished brand $\bar{\phi}$, to be called the "empty brand," its meaning and use will be described later.

The process induced by π/S can be described as a sequence of operations of the following form!

$$q \Leftarrow q_0(q_1 \dots q_k) \quad (k \geq 0) \quad (3.1)$$

where $q_0 \in Q$ is an operator (procedure), and $q_i \in Q$ (for $i \geq 1$) are assumed to be the operands (parameters). Such an operation may have two effects. First, it generates a new (possibly null) *transient object*, denoted here by q , to be called the *product* of the operation. Secondly, it may have "side-effects" on the data base D .

Note that the concepts of *variable* and *assignment*⁽¹⁾ are avoided in this abstraction. Transient objects can be generated and used, but they cannot be changed. The questions of where is a generated object stored, and how is it accessed are ignored at this point.

The specific operations which π/S is allowed to perform are specified by a finite set of *rules* R , to be called *application-rules*. The rules $r \in R$ have the following general structure:

$$r = p \leftarrow (p_0, p_1 \dots p_k) \quad (3.2)$$

where, $p \in B$, $p_i \in B \vee Q_L \vee Q_S$, for $0 \leq i \leq k$.

The right hand side of a rule is called *right*, and will be denoted by p . It is required that every *right* appears at most once in R .

Let us define next the set of all operations which are legalized by a rule r --to be denoted by $op(r)$ --as follows:

Definition: An operation $x_0(x_1 \dots x_n)$ is in $op(r)$ for $r = p \leftarrow p_0(p_1 \dots p_k)$, if the following conditions are satisfied.

- a) $k = n$
- b) For $0 \leq i \leq k$, if $p_i \in Q_L \vee Q_S$ then $x_i = p_i$
- c) For $0 \leq i \leq k$, if $p_i \in B$, then x_i may be any object whose brand is p_i .

It is clear from this definition, and from the definition of R that every

(1) The " \leftarrow " sign in (3.1) is not an "assignment symbol," it is meant only to show that the operation generates an object q .

operation belongs to at most one set $op(r)$. Namely, there is at most one rule which legalizes any given operation.

The existence of a rule $r \in R$ is supposed to have the following affect on every π/S . First, it gives to π/S the right to perform all operations in $op(r)$. Secondly, if any one of these operations is performed, and if it generates a product, this product would be branded by p (the left hand side of r). If, in particular, $p = \bar{\phi}$ (the special null brand), then any product that the operation may have will be "annihilated;" namely, no transient object will be generated. (Instead of the rule $\bar{\phi} \leftarrow (p_0 \dots p_k)$, we will usually write only " $(p_0 \dots p_k)$ ".) Actual techniques for enforcing these rules will be discussed later; for the time being we will just assume that the rules in R are observed.

Note that our rules have nothing to say about the operations themselves; they only specify which operations can be applied to which operands, thus, the name "application rules." The meaning of the operations themselves is supposed to be imbedded in their value parts.

The set R of rules is a union of two sets, R_L and R_S , to be defined as part of L and S respectively. These two sets will be described below in some detail.

3.2 The specification of L and S

As was pointed out before, the environment of π/S is defined jointly by the programming language L , and the schema S . But the two do not play symmetric rules in this definition. The schema will be defined in terms of a particular language L , while L must be completely independent of any schema, because several different schema's may be using the same L . We will now summarize the information needed for the specification of both L and S .

The language L can be characterized within our level of abstraction by the sets Q_L , B_L , R_L .

Q_L - is the set of primitive objects of L . This includes both operators and structures. Specific examples of such primitives will not be given. Here we will only introduce an operator which should be present in every Q_L . It is the *identity* operator I which, when applied to any object, generates a copy of its value part. If the rule $p_2 \leftarrow (I, p_1)$ exists in R , it means that every object branded by p_1 can be copied, and the copy is branded by p_2 . For brevity we will denote the rule $p_2 \leftarrow (I, p_1)$ by $p_2 \leftarrow (p_1)$.

B_L - is a set of *brands*. To distinguish between these brands and those in B_S , we will add the prefix λ to every symbol in B_L . In a typical case, the brands in B_L would closely correspond to the *types* of normal programming languages. We may, for example, have the following brands in B_L : $\lambda.I$ -- to brand integer objects, $\lambda.R$ for real, $\lambda.S$ for string objects, etc.

R_L - is a set of primitive *application rules* of the language L . They have the same general form of rules in R :

$$P \leftarrow (P_0 \dots P_k),$$

but the range of the various arguments is more restricted:

$$p \in B_L, \quad p_i \in B_L \vee Q_L \quad \text{for } i \geq 0$$

As an example, we may have the following rules in R_L :

$$\lambda.I \leftarrow (+, \lambda.I, \lambda.I)$$

$$\lambda.R \leftarrow (+, \lambda.R, \lambda.R)$$

$$\lambda.R \leftarrow (+, \lambda.I, \lambda.R),$$

which means that the *integer* to *integer* addition gives an *integer* result, but the addition of *real* to *real*, or *real* to *integer* gives a *real* result.

It should be pointed out that we do not really expect R_L or Q_L to be explicitly defined in this way. But the brands B_L will be used in the definition of the schema and must be defined explicitly.

The schema S is specified by means of the sets Q_S , B_S , L_S . As was already pointed out, the schema is defined in terms of a given language L . It is connectable to programs written in this language, which belong to a given set of users, called the *patrons* of S .

The set of *permanent-objects* -- Q_S are those objects of the data base which should be accessible to the patrons of S . Two comments are in order here.

a) The designer of the schema may decide that a given object P should not be accessible to the users in its raw form. He may then write a procedure P' which has an access to P , but which manipulates P in a certain desired way. P' , rather than P , is then listed in Q_S , and the users can access P only in a controlled way, via P' . This is an example of access-control, and it is outside the scope of this paper as is the general subject of construction and manipulation of the schema.

b) Our second comment is that the *binding-time* of the symbols in Q_S does not have to be the definition-time of S . It can be the connection-time with a particular program π owned by user U . Thus, the interpretation of any symbol in Q_S may well depend upon U . For example, consider a schema in a medical data base whose patrons are doctors. We may have a symbol PAT in Q_S of this schema which would give to each doctor the set of his own patients. This is a very powerful capability of the schema.

The objects in Q_S cannot be manipulated freely by π/S ; the use of these objects is controlled by the rules in R_S . The effect of these rules on π/S is best seen by means of examples which will be given next.

3.3 Examples

The examples in this section have two objectives: To illustrate the concepts introduced above, and to show their relevance for protection.

Example 1: Consider a binary tree T stored in a data base. Suppose that there are four functions defined on the nodes $n \in T$: $LSON(n)$ and $RSON(n)$ which return pointers to the left and right "sons" of node n ; and $KEY(n)$ and $TEXT(n)$ which return the "data" stored in node n . This data is given in integer and string formats, respectively. (What we call "function" is usually called an "attribute" of a node.)

For every node $n \in T$, let us define $Tree(n)$ to be the subtree of T whose root is n . The following schema gives to every π/S certain access rights to all the nodes in $Tree(N)$ for a given node N .

The schema S is defined by means of the sets Q_S, B_S, R_S :

$$Q_S = \{N, LSON, RSON, KEY, TEXT, STORE\}$$

$$B_S = \{ \bar{n} \}$$

$$R_S = \{ \begin{array}{ll} r_1 : & \bar{n} \leftarrow (N) \\ r_2 : & \bar{n} \leftarrow (LSON, \bar{n}) \\ r_3 : & \bar{n} \leftarrow (RSON, \bar{n}) \\ r_4 : & \overline{\lambda.I} \leftarrow (KEY, \bar{n}) \\ r_5 : & \overline{\lambda.S} \leftarrow (TEXT, \bar{n}) \\ r_6 : & (STORE, TEXT, \bar{n}, \overline{\lambda.I}) \end{array} \}$$

Let us see what can π/S possibly do.

By rules r_1, r_2, r_3 , he can get to the pointers of every node in $Tree(N)$. All such nodes are branded by \bar{n} . Moreover, only nodes of $Tree(N)$ can be \bar{n} -branded because there is no other rule whose product has \bar{n} as its brand. Now, rule r_4 allows π/S to apply the function KEY to any \bar{n} -branded object; that is, to any node in $Tree(N)$, and only to these. Note in particular, that

it will not help the user to get a pointer to a node in the tree T from some outside source; such a pointer will not be \bar{n} -branded. Note also that to get this security there is no need to check at run time if a given argument of KEY is in $Tree(N)$. It is enough that the rules in R_S are satisfied.

Any object generated by the function KEY is branded by $\overline{\lambda.I}$ (rule r4). Since $\overline{\lambda.I}$ belongs to B_L , such an object is under the jurisdiction of the operation rules-- R_L of the language L . Namely, π/S can do with it whatever the language L is able to do with $\overline{\lambda.I}$ branded objects. (In this particular case, $\overline{\lambda.I}$ is assumed to be the brand of integer objects (numbers).) In such a case we will say that an object is released to the user because it is not controlled by the schema anymore. All objects whose brands are from B_L will accordingly be called *free objects*.

Rule r5 is similar to r4; it allows π/S to get the $TEXT$ of nodes in $Tree(N)$. The result is again released to the user in the form of *string* objects (branded by $\overline{\lambda.S}$). Finally, r6 is a right to invoke the procedure $STORE(TEXT, \bar{n}, \overline{\lambda.S})$ which is supposed to store a new value in $TEXT(\bar{n})$; this value must be a free string object.

Let us point out that, as it was explained in section 3.2, the "binding-time" of the various symbols in S is the time of "connection" between π and S . This allows N to depend on the user U . Therefore, the same schema may allow various users to get to different subtrees of T .

Example 2: Consider a medical data base in which there is a set of patients PAT . Each patient $pePAT$ has two attributes, $MEDICAL$ and $PERSONAL$, which hold the medical and personal information about the patient in string format. There are also other functions in the data base which are relevant to us; they will be introduced later.

The *patrons* of the following schema are the doctors in the hospital. Each doctor U is supposed to have access to two sets of patients: MY-PAT is the set of all patients treated by doctor U himself, and DEP-PAT are all the patients in the department in which U works. U would have different access rights to these two sets, as we will see below.

The schema is defined by:

$Q_S = \{MY-PAT, DEP-PAT, NEXT, MEDICAL, PERSONAL, STAT1, STAT2, STAT3, DELETE, ADD\}$

$B_S = \{\overline{q_1}, \overline{q_2}, \overline{q_3}\}$

$R_S = \{ r1 : \overline{q_1} \leftarrow (NEXT, MY-PAT)$

$r2 : \overline{\lambda.S} \leftarrow (\overline{q_1})$

$r3 : \overline{q_2} \leftarrow (NEXT, DEP-PAT)$

$r4 : \overline{\lambda.S} \leftarrow (\overline{q_2})$

$r5 : \overline{\lambda.S} \leftarrow (MEDICAL, \overline{q_1})$

$r6 : \overline{\lambda.S} \leftarrow (PERSONAL, \overline{q_1})$

$r7 : \overline{q_3} \leftarrow (MEDICAL, \overline{q_2})$

$r8 : \overline{\lambda.I} \leftarrow (\{STAT1, STAT2, STAT3\}, \overline{q_3})$

$r9 : (DELETE, MY-PAT, \overline{q_1})$

$r10 : (ADD, MY-PAT, \overline{q_2}) \quad \}$

For simplicity, we assume that the sets MY-PAT and DEP-PAT are ordered sets of names of patients. The function NEXT returns on each call the next patient name. The doctor U is allowed to get the names of patients in these two sets (by rules $r1, r3$), and these names are released as free string objects (rules $r2, r4$). But there are some privileged operators which can be applied only to names retrieved from MY-PAT and DEP-PAT.

By rules $r5$ and $r6$ the doctor can get the medical and personal information about his own patients. Moreover, this information is released to him.

His access to the other patients is more restricted. He cannot get any personal information about these. He can get MEDICAL data about them, but this data is not released; it is branded by $\overline{q_3}$ (rule r7). The doctor cannot, for example, print this information. He can only use it as a parameter to the three functions STAT1, STAT2, STAT3 (Using an obvious convention, rule r8 is equivalent to three rules, one of them, for example, is $\overline{\lambda.I} \leftarrow (STAT1, \overline{q_3})$). These three functions are assumed to be statistical functions built into the data base. The idea is that the medical data of patients not treated by the doctor can be retrieved by him; but he can only use them for some well-defined statistical purposes. Namely, they can only be fed to the three statistical routines.

Finally, the doctor can delete a patient from the set MY-PAT (by r9), or add a patient of the department to MY-PAT (by r10).

3.4 Enforcement of the Application-Rules

Until now it was assumed that the application-rules are honored by the programs π/S ; we will now consider techniques for enforcing them.

The most obvious way for enforcing the rules in R is to do it at "run-time": When an operation $q_0(q_1 \dots q_k)$ is about to be executed, one searches in R for a rule r which gives the *right* to execute this operation. Once the operation is carried out, and if it generates a *product*, this product is branded by the left hand side of the rule r .

Because of obvious reasons of efficiency, run-time checking should be avoided whenever possible. For this purpose in mind, let us introduce into our model the concepts of *variable* and *assignment operator* which were avoided until now.

Definition: A *variable* is an object which has another object as its value. Moreover, the *value part* of a *variable* can be changed. The only way to change the value of a variable is by means of the operator ":", to be called *assignment operator*, as follows: If *v* is a variable and *q* is a non-variable object, then the operation $:(v,q)$, which is conventionally denoted by $v:=q$, copies object *q* into the value part of *v*. (For simplicity we assume that the product of the assignment operator is null.)

To fully incorporate variables into our model, we introduce the following convention. Whenever a variable appears in an operation, other than as the first argument of the assignment operator, the effect is as if the value part of the variable appears in the same place. This, of course, is the normal treatment of variables.

Now, it makes sense to talk about the *scope* of a particular variable which is defined here to be the set of objects which can legally be assigned to it. The *scope* of a variable, being so defined, obviously depends upon the set of rules R . For illustration, consider the following example.

Let a language L be defined by:

$$Q_L = \{\text{DECLARE, INTEGER, :=, ...}\}$$
$$B_L = \{\overline{I}, \overline{IV}, \dots\}$$
$$R_1 = \{r1 : \quad \overline{IV} \leftarrow (\text{DECLARE}, \text{INTEGER})$$
$$r2 : \quad (:=, \overline{IV}, \overline{I})$$

By rule r1, the operation `DECLARE(INTEGER)` creates an object branded by $\overline{\text{IV}}$. Suppose that this is the only operation which generates $\overline{\text{IV}}$ -branded objects. By rule r2, $\overline{\text{IV}}$ -branded objects are *variables*. If we also assume that r2 is the

only rule in R in which the symbols ":@" and " \overline{IV} " appear as the first two parameters, then it is clear that only integer objects can be assigned to IV -branded objects. These objects are, therefore, what is normally called "integer variables."

In general, for every brand β , it is possible to generate variables whose scope is the set of all objects branded by β . We will say that the *scope* of such a variable is β .

Now, if the scope of every variable used in a program π is some $\beta \in B$, then we do not need explicit brands to be stored in our objects. Moreover, if the names of all variables in a program are declared together with their scope, and if their names are used explicitly in the program, then it is clear that the application rules can be enforced at *compile-time*.

3.5 On the Concept of "Type"

The reader may be wondering why did we choose to use the names *brand* and *scope* for what may seem to be simply the standard concept of *type*. We will make a small digression from our main subject to clarify this point.

First, it should be pointed out that there is actually no *standard* meaning to the term "type" in programming. This concept is currently in a process of evolution, and different people may have different things in mind when they use the term "type." This is one good reason for avoiding the use of this term. But there is also another, more serious reason: The term "type" is frequently used for two fundamentally different concepts at the same time.

First, types are used as descriptors of objects. For example, the statement "the value of a function is of *type* integer" is a statement about the objects generated by the function. (In this paper we used *brands* for this purpose.) However, the term *type* is also used in an entirely different way: the statement "the *type* of a variable is INTEGER" is a statement about the

scope of the variable. In a language like FORTRAN or ALGOL it may not be clear that there are two different concepts here, because in these languages the *scope* of a variable must coincide with a set of all object of a given type (or: with a given brand), e.g., *integer variable*, *real variable*, etc. The difference between the *scope* of a variable and the *brand* of an object becomes important when the *scope* of a variable may be the set {1 ... 10} for example. Thus, there should be no correspondence between the first interpretation of the type and the second interpretation - as a *scope*.

The failure to make a clear distinction between these two concepts by using the same term like "type" or "mode" for both of them may cause considerable confusion. Like the "mode-UNION" confusion in ALGOL/68.

It should be pointed out that if one wishes to use an axiomatic approach to the concept of type, then the *scope* of a variable is not a new concept at all. It is a property of a *variable-object* which is derivable from the underlying application-rules of the language, similarly to what was done in the last section.

4. Extension of the Basic Model

In this section we will see that our model is too primitive to be practical. Several extensions to it will be considered but will not be discussed in depth.

4.1 Value-Dependent Rules

The most obvious limitation of our rules is that they are defined in terms of the names in Q , and the brands in B only; and they are totally invariant of the value parts of the objects involved. For example, rule r_5 in example 2 of section 3.3 allows π/S to get the medical information about any patient whose identifier is branded by \bar{q}_1 . However, one might want to condition the application of this rule on some property of the patient. For that, let us generalize the rules defined in (3.2) to the form:

$$r' : p \leftarrow (p_0 \dots p_k)/C(q_0 \dots q_k), \quad (4.1)$$

The meaning of this rule is the following: suppose that $q \leq q_0(q_1 \dots q_k)$ is an operation which is about to be performed by π/S , and that this operation belongs to $op(r)$ for the rule $r : p \leftarrow (p_0 \dots p_k)$. π/S will be permitted to carry out the operation only if $C(q_0 \dots q_k)$ is satisfied (C is assumed to be a predicate).

Another restriction that one would like to lift from our rules is the fact that all the operations, $op(r)$, legalized by the rule r , generate objects with the same *brand*. A higher "resolution power" of A brands can be achieved by the following form of rules:

$$r'' : (p^1/c^1, p^2/c^2, \dots p^n/c^n) \leftarrow (p_0 \dots p_k)/C(q_0 \dots q_k). \quad (4.2)$$

The right hand side of r'' is identical to r' . The p^i in the left side of r'' are brands, and c^i are predicates defined over $q, q_0 \dots q_k$ of the operation in hand. The left hand side of r'' determines the brand of the product

of the operation legalized by it, as follows: The predicates c^i are computed one by one beginning with c^1 . If c^i is the first to be satisfied, the brand of the product would be p^i . If none is satisfied, then the brand is ϕ .

4.2 Variable Brands

The *brands* defined in section 3 were a finite set of symbols. These symbols appeared explicitly in the application rules. There seem to be good reasons, however, to use more complex structures as brands, and to have variable brands appear in our rules. The following example indicates such a need:

Consider a medical data base and a schema S which is supposed to be used by nurses in the hospital. The following is a partial list of the rules in R_S given in the form of section 3. We will first explain these rules, and then point out to the difficulty that they present.

r1 : $\bar{d} \leftarrow (\text{NEXT}, \text{DOC})$
r2 : $\overline{\text{pset}} \leftarrow (\text{PAT}, \bar{d})$
r3 : $\bar{p} \leftarrow (\text{NEXT}, \overline{\text{pset}})$
r4 : $\overline{\text{pp}} \leftarrow (\text{PERSONAL}, \bar{p})$
.
.
.
r5 : $(\text{SEND}, \overline{\text{pp}}, \bar{d})$

DOC is a set of names of doctors which is accessible to a nurse U . Each doctor name generated by $\text{NEXT}(\text{DOC})$ is branded by \bar{d} (rule r1). The function PAT applied to a legal name of a doctor returns a set of patients of this doctor, to be branded by $\overline{\text{pset}}$ (r2). There may be many such sets; U can get the names in each of them by the operator NEXT; the patient name so received is branded by \bar{p} . By rule r5, U can get the personal information of every \bar{p} -branded patient.

Now suppose that the operation $\text{SEND}(\text{text}, \text{doct})$ sends the information *text* to a file which belongs to the doctor *doct*. We would like to allow *U* to send personal information about a patient to the file of the doctor who treats this patient, and only to him. Rule r_5 , however, does not do that; it allows *U* to send such information to any doctor. Moreover, there does not seem to be any clear way to impose such a restriction by our form of rules. To do so, we will have to generalize our brand concept:

Suppose that every *brand* is a pair $\overline{\beta_1 \beta_2}$; where β_1 is a symbol, such as we had until now; and β_2 is an arbitrary integer number. Using such brands we redefine our rules as follows:

- $r_1' : \overline{d.i} \leftarrow (\text{NEXT}, \text{DOC})$
- $r_2' : \overline{\text{pset}.i} \leftarrow (\text{PAT}, \overline{d.i})$
- $r_3' : \overline{p.i} \leftarrow (\text{NEXT}, \overline{\text{pset}.i})$
- $r_4' : \overline{\text{pp}.i} \leftarrow (\text{PERSONAL}, \overline{p.i})$
- $r_5' : (\text{SEND}, \overline{\text{pp}.i}, \overline{d.i})$

i is assumed to be a variable which may hold any integer. It is assumed that every activation of $\text{NEXT}(\text{DOC})$ which retrieves a doctor's name, brands it by $\overline{d.i}$, with a different *i* for every doctor. This *i* is carried to the brands of the patients (by rules r_2' , r_3'), and their personal information (rule r_4'). Thus, the brand of the personal information of a patient carries in it the identification of the doctor of that patient. It is therefore possible, by rule r_5' , to allow *U* to send such information to the right doctor only.

This example clearly suggests that variable, and non-scaler brands may be useful. A systematic model of such brands is yet to be worked out.

4.3 Global Brands

The *brands*, as they were introduced in section 3, are strictly local to the schema and to the users' programs working under it. An obvious generalization to make is to let objects in the data base have global brands; indeed, such a generalization is absolutely necessary. The global brands may be used not only by rules in individual schemas, they may also be utilized in the formulation of a "constitution" for the data base, namely, a set of rules with which every schema must conform.

5. Conclusion

Two aspects of security in data bases were discussed in this paper: the description of the *security state*, and its *enforcement mechanism*. The conventional approach to these two aspects has been found to be inadequate, and a new approach has been proposed. According to the proposed approach, the user's program- π which interacts with the data base operates under the jurisdiction of a set of rules, collectively called *schema*, which in effect defines the security state of the data base with respect to a given set of users. The function of the schema is to authorize the operations performed by π . In such a way it can control the fate of information originated by the data base, or otherwise related to it, even when such information is in the user's working space. But the schema has no authority over information which is not related to the data base, and over the computation performed by π on such information.

This paper is only a first step in a research which should progress in several directions, as is briefly outlined below:

a) The model for user data base interaction proposed here is a radical change from the conventional form of such interaction under which users' programs were completely free. Various pragmatic aspects of our approach should therefore be carefully assessed. In particular, it is not quite clear how various general purpose languages should be adapted for the role designed for them in our model. (It should be pointed out that the need to change general purpose languages in order to adapt them for the interaction with data bases is not new with us, see, for example, DBTG report [Dbtg.71]).

b) A convenient language for schema specification should be developed.

c) The model proposed here is very basic. Some possible extensions of it were briefly outlined in section 4; they should be pursued further.

d) Most important of all, it should be realized that controlled interaction between users and the data base is only a necessary condition for its security; it is by no means sufficient. There are many more problems to be solved before one gets a (reasonably) secure data base system. For example, methods for generation and manipulation of schemas must be formulated.

References:

- [Bro.71]: P. S. Browne and D. Sheinauer, "A Model for Access Control," Proceedings of the ACM SIGFIDET Workshop, 1971.
- [Con.72]: R. W. Conway, W. L. Maxwell, H. L. Morgan, "On the Implementation of Security Measures in Information Systems," CACM, April 1972.
- [Dbt.71]: DBTG report to the CODASYL Programming Language Committee
- [Grh.72]: G. S. Graham, P. J. Denning, "Protection-Principles and Practice," AFIPS/SJCC 1972.
- [Jon.73]: Anita K. Jones, "Protection in Programming System," Thesis, Carnegie Mellon University, June 1973.
- [Lam.71]: B. W. Lampson, "Protection," Proc. Fifth Annual Princeton Conference on Information Sciences and Systems, March 1971.
- [Owe.71]: R. C. Owens, Jr., "Evaluation of Access Authorization Characteristics of Derived Data Sets," Proceedings of the ACM SIGFIDET Workshop, 1971.

SOSAP-TR-8

April 1974

COMMENTS ON PRIVACY OF DATA BASES

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

Abstract

The protection of privacy of data bases is discussed. The main results of the paper are: a) The conventional *access-matrix* model, frequently used for the specification of privacy measures, is found not to be suitable for data bases. b) The *access control* mechanism commonly used to enforce the privacy measures of data bases is found not to be powerful enough, since there are privacy measures which cannot be enforced by it. The paper argues that it is necessary to impose restrictions on the user's program which interacts with the data base.

Introduction

In spite of the special importance of privacy in data bases, most of the theoretical work on privacy and security was performed in the context of operating systems. Some of the principles developed in these studies were then adopted for application in data bases, without due regard to the fundamental differences between these two subject matters. An example of such an adoption is the *access-matrix* model which was developed for operating systems [1,2], and was then adopted as the conceptual framework for protection of data bases as well, (see [3] for example).

One of the conclusions of this paper would be, however, that in spite of the apparent success of the *access-matrix* model in operating systems, it is not a suitable model for data bases. The point that we are trying to make is that the subject of protection in data bases has some unique aspects which deserve an independent study. Some of these aspects will be discussed in this paper.

As is implied by the title, there is no intention to conduct a comprehensive study of privacy protection in this paper, we will be just making several comments. In these comments we will address ourselves to two aspects of privacy protection. First, the specification of the "privacy state". Namely, how does one specify what it is that a given user is allowed to do to the data base, and which information can he get from it. The second aspect which will concern us is, the *enforcement* of the privacy-state.

1. On the Specification of Privacy States

The prevailing approach to the specification of the privacy state of a system is essentially the following: Assuming that there is a finite number of distinguishable objects in the system, and that there is a finite set of potential users (or classes of them), all we have to do is to specify which operations can every user apply to every object. One way to represent that is by means of a matrix, typically called *access-matrix*. The element A_{ij} of such a matrix would be the list of operations, or *accesses*, which user i is allowed to apply to object j . An equivalent model, which is frequently more convenient, is the *environment* or *capability model*. According to this model, every user who interacts with the system operates within an *environment* E which is essentially a set of *rights*. Every right in E is a pair (α, q) , where q is an object in the system and α is an access made, or a name of an operation. The pair (α, q) serves as a right, for the user operating within the environment E , to apply α to q .⁽¹⁾

Although this model for privacy specification was initially introduced for operating systems [1,2], it was adopted by many researchers as a conceptual framework for the protection of data bases as well, (see [3], for example). This is unfortunate, however, because as we will show below the *access matrix* model, or the equivalent *capability model*, is not general enough, and is particularly inadequate for protection of data bases. First, let us consider an example.

(1) This model can be generalized, in order to cope with "value dependent" privacy measures, as follows: The *right* (α, q) can be conditioned on a predicate, which may be an arbitrarily complex procedure.

Let $\text{EXCH}(x,y)$ be an operator which exchanges its two arguments, provided that they are of the same type t (whatever such an "exchange" may mean). Let a, a', b, b' be objects of type t , and let U be a user who is allowed to exchange a with a' and b with b' , but has no right to manipulate these objects in any other way. It is quite obvious that this privacy measure cannot be specified by means of the formalism presented above. The best one can do is to allow U to apply the operator EXCH to all four objects, by providing him with the "rights": $(\text{EXCH}, a), (\text{EXCH}, a'), (\text{EXCH}, b), (\text{EXCH}, b')$. But this is too permissive, it would, for example, allow U to exchange a with b .

More generally the *privacy state* of a system can be specified by means of an *access matrix* if the primitives available to the user, for interaction with the system, are all operations on single objects. As it is usually the case in operating systems, when these primitives can be considered to be the various *access modes* like "read access", "write access", etc. However, as was illustrated by the EXCH example, the access matrix model is not powerful enough to deal with primitives which operate on a number of objects. To specify that a user is allowed to apply an operation p to the objects q_1, \dots, q_n , we should use something like a tuple (p, q_1, \dots, q_n) rather than the pair (α, q) of the capability model. For instance, the privacy state of our example might be specified by the following two triples:

$(\text{EXCH}, a, a'), (\text{EXCH}, b, b')$.

Now, the point is that the primitives operations which are available to the user of a data base are sometimes high level procedures which operate on several objects; indeed, it has been shown in [5] that such

primitives may be essential for the *integrity* of the data base. Therefore, the *access-matrix* is not a suitable tool for specification of privacy in data bases. (The use of n-tuples for that purpose was discussed in [6]).

2. A Model for User-database Interaction

In this section we will describe a schematic model of the process of user-database interaction, which will serve as a framework for the subsequent discussion of enforcement of privacy in data bases.

Let D be a data base, and U one of its users. According to the Data Base Task Group report [4] the user should not "see" the data base D itself, but an abstract image D' of it. D' would ideally contain only those parts of D which are relevant to U , and in a form most suitable for him. The definition of such an image is usually called *sub-schema* and will be denoted by Σ . In order to interact with the data base the user must first *connect* himself to a certain Σ , (there would, typically, be many of them in a given data base), such connection would generate the environment in which the user's program would operate. We will denote the user's program by Π , or $\Pi(u)$.

There are many issues which must be discussed in order to turn the above into a meaningful model for user-database interaction. At this point we will consider just one of them: the primitives operations which should be available to the user for the interaction with the data base. According to the conventional approach, that of the DBTG[4], in particular, these primitives are essentially *update and retrieval operations*; the same kind of operations which are traditionally used to manipulate files. However, data bases are not files, and update and retrieval operations are not suitable to manipulate them. One reason for that, which was discussed in [5], is that in order to maintain the privacy and integrity of the data base, the user must be provided with higher level

operators which manipulate the data base in some predesigned, presumably correct, way. These operators, which will be called *D-operator*, or *D-function*, have a central role in the formation of the *abstract image* of the data base which should be accessible by the user. The D-operations should, therefore, be specified in the sub-schema, (see also figure 1). In general, a D-operator may have two types of effects: It may affect the data base itself, and it may generate information into the storage space of the user's program which invoked it, such information will be called the *outcome* of the D-operation. In other words, the *outcome* of a D-operator is the information retrieved from the data base.

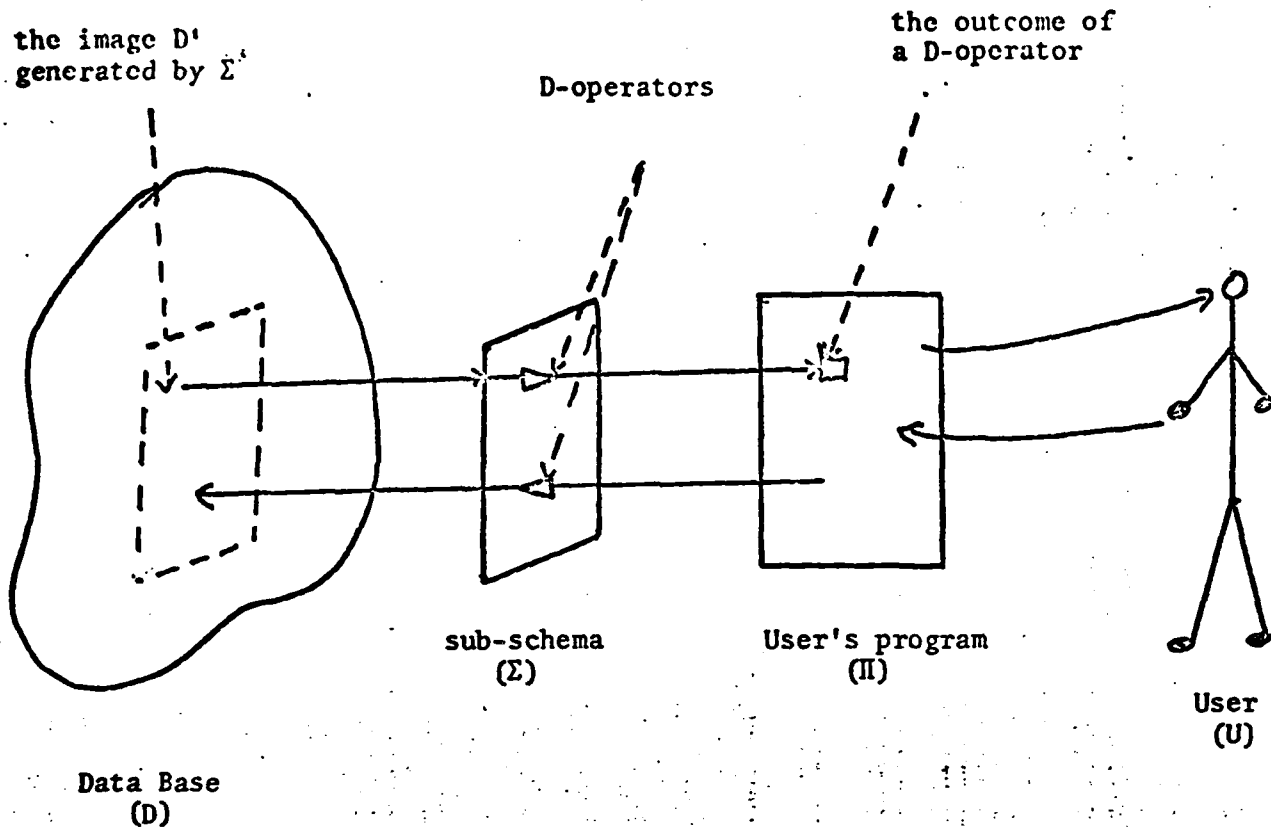


Figure 1: A schematic model for user-Database interaction

3. On the Enforcement of Privacy

The conventional approach to interaction with data bases, as outlined above, provides a basic framework for privacy protection, which can be viewed as a two-level mechanism: First there are privacy restrictions as to which sub-schemas can be connected to the programs of a given user, and there should be a mechanism which enforces these restrictions. The rest is up to the sub-schema itself. It is the role of the sub-schema as a privacy enforcer which will concern us here.

According to the conventional approach, that of the DBTG [4] in particular, the sub-schema acts essentially as an *access-control* mechanism: It has an authority to decide which parts of the data base can be accessed by Π , and which operations can be applied to the data base, but it has no authority over what happens inside the user's program itself. In particular, once a piece of information is transmitted into the storage space of Π , it is automatically released from the control of the data base and of its privacy rules.

This may seem to be the natural approach to protection, indeed, the term "access control" is frequently used as a synonym for "protection". But as we will show in this paper, the access-control mechanism has some inherent limitations, and cannot be used as a universal privacy enforcer. We will see that some privacy measures can be enforced only by imposing restrictions on what happens inside the user's program, or are much easier to enforce with such a control over Π .

There are, essentially, two problems with the access control mechanism as the privacy enforcer; one of them will be treated in the next section and the other is introduced below by an example:

Example 1

Consider a medical data base D , and let P be a set of patient's names in it. Let $A_1 \dots A_k$ be D -functions defined over the members of P . (A_i are "functions" in the set theoretical sense of the term, one can also view $A_i(q)$, for $q \in P$, as the i -th attribute of q). Let the user U be a doctor, and let $P'(U)$ be the set of all patients of doctor U , ($P'(U)$ is a subset of P). Let ρ be the following privacy rule: " U should have no access to patient names which do not belong to $P'(U)$, nor to the attributes of such patients". How can this privacy rule be enforced?

First one may try to enforce the rule ρ simply by providing U with the partial image of the data base which contains the names of his patients only. This can be done as follows. Let $MYPAT$ be a D -function provided to U by the sub-schema to which he is attached. Suppose that $MYPAT$, when invoked by $\Pi(U)$, returns as its outcome the next name from $P'(U)$, (whatever "next" may mean). Assuming that this is the only way for U to retrieve elements from P , the rest of our privacy rule may seem to be automatically satisfied. Namely, a request to retrieve $A_i(q)$ can be honoured without further checking, because q can be nothing but a member of $P'(U)$. Unfortunately, however, this is not the case. Even if U can get only names of his own patients from the data base, he may still be able to get names of other patients from an outside source, or, he may simply "invent" a name. Thus, U has the potential ability to invoke $A_i(q)$ for $q \notin P'(U)$.

From the above it seems that the only way to guarantee that U does not get the attributes of a patient $q \notin P'(U)$, is to check if q is

actually a member of $P'(U)$ every time that $A_i(q)$ is invoked. This, however, is a very wasteful solution, it means for example that the membership of q in $P'(U)$ may have to be checked several times for the same q .

To understand the nature of our problem let us look at it from a more general point of view.

Let F be a D-function, and v a variable in the storage space of Π . Suppose that F is invoked by Π and that the outcome of F is stored in v . One can say that the variable v carries two types of information in it: First there is the set of bits actually stored in v , which we will call the explicit information in v . In addition, v carries an important implicit information, which is *the fact that the content of v was generated by the D-function F* ; in a sense, this *implicit information* provides the interpretation for the *explicit information* stored in v . For instance, in our example above, the outcome of MYPAT is not just a string of bits, it is known to be *the name of one of the patients of U* . The existence of such implicit information is so natural and obvious, that one hardly gives any thought to it. However, consider the following situation.

Let G be another D-function in our data base, which admits one parameter. Suppose that G is invoked by Π , with v as a parameter. The invocation $G(v)$ obviously communicates to the function G the *explicit information* in v , that is what parameters are for. But the *implicit information*, which gives to v its interpretation, is not communicated to G ; because G has no way of knowing how was the content of v generated. Coming back to example 1, if $A_i(q)$ was invoked by Π , there is no way for the

function A_i to know that the content of q is in fact "the name of a patient of U " without actually checking it. Namely, for the function A_i the parameter q is nothing but a string of bits, its *implicit information* is lost.

To summarize; the *implicit information* carried by the outcome of a D-function into the storage space of Π , cannot be carried back into the data base. This loss of information, which as we saw may cause difficulties in the enforcement of certain privacy rules, is due to the complete freedom that Π has in the manipulation of its own storage space. As we will show next, one can maintain the credibility of such implicit information by imposing restrictions on what the user's program can do with information retrieved from the data base.

Consider the following modifications, borrowed from [6], to the conventional model of interaction with data-bases. Suppose that the outcome of a D-operation, when transmitted to the storage space of Π , can be marked, or *branded*, by one of a set of *brands* specified in the sub-schema. (This branding process should be under the complete control of the sub-schema, in a manner not to be specified here). The branded information-objects, although they are stored in the storage space of Π , cannot be manipulated freely by it. Rather, they are subject to a set of rules, to be called D-rules, provided by the sub-schema which in effect, specify the operations which can be applied to the branded objects. Every *brand* which appears in the sub-schema, also *induces* a new "type" into Π , in the following sense: If β is a brand in Σ , then one may declare, in Π , a variable to be of "type β " much in the same way as one declares variables to be of "type integer", say. A variable of type β may contain only β -branded information objects.

Coming back again to example 1: Suppose that the outcome of MYPAT is branded by the brand β . Suppose also that the D-rules provided by Σ allow Π to use β -branded objects non-destructively, but do not allow it to modify them in any way. Under these assumptions the privacy restriction ρ can be enforced simply by requiring that only variable of "type β " would be used as parameters of the D-functions A_i . Because, due to the restrictions above, a variable of type β can contain only names of patients of U . We may say that the restrictions on the manipulation of β -branded objects preserves the implicit information in them.

Thus, we saw that certain privacy measures can be enforced more efficiently if the user program can be controlled by the data base. We will now see that for a certain class of privacy restriction such control is essential.

4. "Intentional Resolution Power" of Privacy Protection

One of the most important characteristics of privacy protection mechanisms is the fineness in which the privacy measures can be specified, which may be called the *resolution power* of the privacy protection. One may be interested in *resolution* along two orthogonal "dimensions". One of them, whose importance is usually well appreciated, is the specification of which parts of the data base should be revealed to the user; the other dimension, which will be called *intentional resolution*, is related to "what is the user allowed to do with a piece of information revealed to him". It is this second aspect of the resolution power of privacy protection which will concern us here.

It may seem that *intentional resolution* is mainly a legal matter, not within the realm of computer technology, because once a piece of information is revealed to a human user, there is no way for the data base system to impose restrictions on how it is used. Indeed, in the traditional approach to privacy protection, the privacy enforcer is viewed as a kind of mediator between the data base and the user, whose job is to decide which information can be revealed to the user. There is no place for *intentional resolution* under this approach. Actually, however, the data base does not usually communicate information directly to the human user, but to his program. Moreover, much of the information retrieved from the data base is needed only for a certain computation to be performed by the user's program, and there is no need for the user himself to see it. To comply with the "need to know" principle, such information should be released to the program Π , but Π should be prevented from revealing

this information to the human user. There is a difficulty here, however; under the conventional approach, the data base, or its privacy enforcer, has no more control over the user's program, than it has over the human user himself. In particular, once a piece of information is transmitted to Π there is no way, short of a manual audit of the program, to prevent Π from printing this information, for the user U to see.

It is clear, therefore, that in order to provide any degree of intentional resolution, the privacy enforcer must be able to impose restrictions on the computations performed inside the user's program, and not only on the D-operators invoked by it. This is again the conclusion of section 3, derived in a different way. The implementation of intentional resolution is discussed in some detail in [7]. This section will be concluded by two examples which are intended to demonstrate the practical importance of intentional resolution.

Example 2

Consider a data base D , and a highly confidential set of records, $F = \{f\}$, in it. Let U be a programmer who is commissioned to apply a transformation T to every record $f \in F$ and to store the transformed records back into the data-base, as a set F' . Suppose that due to the confidentiality of F we do not want its content to be revealed to the programmer U himself, or to leak in any form or shape to the outside world. In other words, we want the programmer *to eat the cake, but not to have it in his stomach*.

This case represents an important class of privacy problems. It is this author's belief that the most serious threat to the privacy of data bases does not come from outsiders who are using ingenious techniques to penetrate the system, but from insiders who have the data base at their fingertips. Every data-base system employs "maintenance programmers" whose

job is to perform routine transformations and manipulations of various files. Most of these programmers have no need to actually see all the information manipulated by them, and they should be prevented from doing so.

Example 3

Consider a medical data base D. Let $PAT = \{p\}$ be a set of patient names in D. Let $T = \{t\}$ be a set of treatment codes, and let $FEE(t)$ be a function, in the set-theoretical sense, which gives the fee, in dollars, that has to be paid for treatment t . Suppose also that for every patient p there is a list, $TLIST(p)$, of all the treatments received by p .

Let U be a clerk who has to write a program to compute the charge $c(p)$ of patient p , in order to send him his bill. $c(p)$ is given by:

$$c(p) = \sum_{t \in TLIST(p)} FEE(t)$$

It is clear the U must have an access to the list of treatments received by p , in order to compute $c(p)$. At the same time, however, $TLIST(p)$ is a confidential information which the clerk has no business seeing himself, nor should he be able to leak it to anybody else.

Thus $TLIST(p)$ should be revealed to a program written by our clerk, but for a limited purpose only.

For a discussion of the realization of the intentional resolution required in these two examples, the reader is referred to [7].

Conclusion

Two issues of privacy protection in data bases were discussed. First, it has been shown that the conventional *access-matrix* technique for the specification of the "privacy state" of a system is not as general as it is sometimes believed to be, and it is particularly unsuitable for data bases. A simple generalization of the matrix model was proposed.

The second issue has to do with the enforcement of privacy. It has been shown that not all privacy measures can be enforced by *access-control* alone, and that it is necessary for the privacy enforcer to have some control over the user program itself. The realization of such a control was not discussed in this paper, but it is safe to say that it would require some radical changes in the conventional approach to the user-database interaction. The subject was discussed further in [5,6,7], but there is much more to be done.

SOSAP-TR-10

July 11, 1974

ON THE FORMATION OF ABSTRACT DATA TYPES

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAH15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

One of the most elegant and powerful concepts, recently developed for programming languages is the concept of *abstract data type*, it seems to have been invented by the designers of SIMULA language [1], and received its final touch (to date) in a paper by Liskov and Zilles [2] (we will refer to this paper by LIZ). *Abstract data type* was defined by LIZ as "a class of objects which is completely characterized by the operations which may be performed on those objects." To implement this concept they introduced a new linguistic construct called *operation cluster*. The cluster serves as a structural template for the objects of the type *t* being defined by it. In addition, the cluster contains the definition of a set of procedures which are to serve as the *characteristic operators* of type *t*. That is to say, an object of type *t* can be manipulated only by means of these operators.

The usefulness of the cluster concept is unquestionable. In addition to being an important tool for well structured programming, it provides the programmer with the very desirable capability of imposing restrictions on his own program. But some of the claims made by Liskov and Zilles about their cluster concept are not completely justified. The following is a quotation from their paper: "We believe that the above concept captures the fundamental properties of abstract objects", since, they say: "The behavior of an object is captured by the set of [its] characterizing operations." One objection to this rather strong claim rests on the following observation:

The behavior of a data object cannot always be fully characterized by the operators which are applicable to this object alone. One might have to take into account the possible interactions of the given object with other objects, which may belong to the same, or to different classes. The difficulty here is that an interaction between two or more objects, cannot always be reduced to a sequence of legal operations on the objects which participate in the interaction. Therefore, as we will see more specifically below, an interaction between objects cannot always be implemented within the framework proposed by LIZ. An example may clarify this point.

Let JOB and PERSON be two classes of objects (abstract data types), and let the objects j and p be instances of these classes, (j and p may be also viewed as "identifiers of", or "pointers to" the corresponding objects). Let R be a one-to-one relation between the two classes, which specifies "who is appointed to which job". One way to implement such a relation is the following: Let $j.person$ and $p.job$ be components of the objects j and p , such that $j.person$ points to the PERSON object assigned to j , and $p.job$ is defined symmetrically. Now suppose that we have to impose a certain discipline on the appointments. According to the philosophy advanced by LIZ, the following technique is implied: The clusters which define the classes JOB and PERSON should not provide any operators for direct update of the components $j.person$ and $p.job$ (namely, from outside of these clusters, $j.person$ and $p.job$ would be "read only" variables). Instead, there should be an operator *appoint* (p, j) which performs the appointment of person p to job j , under the discipline at hand. Thus the behavior of objects of types JOB and PERSON cannot be fully captured just by the operators defined on these objects alone, one must consider also the operator *appoint* which is, in effect, an interaction between a (j, p) pair. Moreover, under these conditions the operator *appoint* cannot be implemented in a language based on the cluster principle. That is so because the operator *appoint* has to change the components $j.person$ and $p.job$ of j and p . But the first can be changed only within the cluster which defines JOB, while the second can be modified only within the PERSON cluster. Thus there is simply no environment in which *appoint* can operate.

Two solutions to this problem suggest themselves: First, one can remove the relation R from the body of the objects j and p ; Instead of using the components $j.person$ and $p.job$, one can define R as a set of pairs $\{(j, p)\}$, using the mathematical definition of "relation". To manipulate this relation one does not need any privileged access to the objects themselves since only the pointers to the objects are involved. Thus, the operator *appoint* can be defined outside of the clusters of JOB and PERSON. However, such decentralization of the data which is relevant to the objects at hand is not compatible with the underlying philosophy advanced by LIZ. (Incidentally, the above suggests that

in order to define the behavior of a class of objects, one may have to specify where pointers to these objects can be stored. The characteristic operators of objects, and even the possible interactions between objects may not be enough.)

The second solution to our problem calls for more sophisticated scope rules than those suggested in LIZ. For example the procedure *appoint* can be implemented if it is possible to give it a privileged access to both *j.person* and *p.job*, without giving such an access to the rest of the program.

In spite of its limitations the cluster may be an ideal construct for most programming languages. Considering its simplicity, it accomplishes amazingly much so that it can be rated as a "best buy". However, the limitation of the cluster concept becomes crucial in a programming environment where security is essential, such as the construction of operating systems and data bases. It is interesting to point out in this context that studies of protection in programming systems tend to do the same oversimplifications made in LIZ. For example, Lampson's protection theory [3] specifies the "protection state" of a system by listing all the accesses (operators) which are applicable to every object. This again ignores interactions between objects and is not, therefore, general enough. (This problem, in the context of data bases, was discussed by the author in [4].) The paper of Morris [5] which studies protection in programming languages seems to suffer from the same problem.

References

- 1) Dahl, O. J., Myhrhand, D. and Nygaard, D., "The Simula 67 Common Base Language" publication S-22, Norwegian Computing Center, Oslo 1970.
- 2) Liskov, B. and Zilles, S., "Programming with abstract data types", Symposium on very high level languages, March 1974.
- 3) Lampson, B. W., "Protection", Proc. Fifth Annual Princeton Conference on inf. sci. and systems, March 1971.
- 4) Minsky, N., "Comments on privacy of data bases", Somp. Sci. Dept., Rutgers University, SOSAP-TR-8, April 1974.
- 5) Morris, J. H. Jr., "Protection in Programming Languages", CACM, January 1973.

SOSAP-TR-11

September 1974

PROTECTION OF DATA-BASES, AND THE PROCESS OF USER DATA-BASE INTERACTION

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHCl5-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

1: Introduction

In spite of the significant variations among existing approaches to protection of computing systems, they all seem to share the following principle: If D is a system to be protected from undesirable interaction with the outside world, and Π is a process of computation which interacts with D; then *all that one has to do in order to protect D, is to guarantee that Π applies only "legal operations" to D*. This statement is based, of course, on the assumption that the system itself is "correct", and that for every process Π , there is a well-defined set of "legal operations", with respect to Π . A direct consequence of this is that in order to protect D it is enough to monitor and validate all operations directly applied to it, while the internal behavior of the process which invokes these operations seems to be irrelevant for the protection of D. Thus, Π can be viewed as a "black box" by the protection mechanism of D.

This general approach, usually called "protection by *access-control*", is being used successfully for the protection of operating systems. Also, some attempts have been made lately to provide general purpose programming languages with certain access-control capabilities, in order to increase the reliability of programming. Notable are the *encapsulation* technique of Liskov and Zilles [4], and the *sealing* technique of Morris [10]; both are bases, to some extent, on access-control type of protection.

In spite of this success of the access-control mechanism, and its intuitive appeal, it turns out not to be sufficient, at least not for data bases. We will show in this paper (section 2) that the access-control mechanism has some inherent limitations: there is a definite limit to the properties which can be protected by it, in addition this mechanism is inherently inefficient, in certain respects. It will be suggested, that in addition to access-control, data bases should be protected by imposing restrictions on the internal behavior

of the user's programs which interact with them. A new approach to the process of user-database interaction, which facilitate such a control over the user's programs will be developed (section 3), and its implication to protection will be demonstrated by several examples (section 4).

Our discussion will be virtually independent of the structure of the data base itself. Yet, the results of this paper have some definite implications to the architecture of data bases, these implications will be mentioned briefly in section 6.

2: The process of user-database interaction, and its problematic?

2.1: What is data base?

We do not really intend to answer this question, except for saying that data base would be viewed here not as just a collection of data items; but as the data, together with rules (or assertions) about it. These rules can be broadly classified into: a) *structural rules* which assert what is the proper state of the DB⁽¹⁾ in any given moment in time; and b) *behavioral rules*, which state how should the DB behave under interaction with the outside world, (in particular, this class includes the so called *privacy measures* or *restrictions*). It is clear that only the existence, and credibility of such rules provide a meaning to the raw data stored in the DB. It is also clear that it is not enough for such rules to be only in the mind of the users of the DB, nor can we rely on the good will of the users to abide by them. It is therefore mandatory for these rules, and for any mechanism which is necessary for their enforcement, to be an integral part of the DB. The fact that most existing data base systems have hardly any way even for stating rules about the data, not to mention mechanisms for their enforcement, is no indication for lack of importance. It only suggests that the enforcement of such rules, is a very difficult proposition. In this paper we will assume that the data base does contain a set of rules about its own structure and behavior, to be called *D-rules*. The enforcement of these rules is what we will call *protection* of the DB.

Since protection is indeed a difficult subject, which cannot be treated in a single paper, we will restrict ourselves only to those aspect of protection which are directly relevant to the process of interaction.

(1) We will frequently use the abbreviation DB for the term "data base".

between a known user and the DB, (to be called the process of *U-D interaction*, for short). We will begin by presenting, schematically, the conventional approach to this process.

2.2: The conventional approach to U-D interaction - A schematic model

There are four main participants in the process of user-database interaction, (see figure 2.1).

They are:

- a) The *data base* - D, whose structure is left unspecified;
- b) the *user* - U, whose identity is assumed to be known;
- c) the *user's program* - Π , which may be a traditional kind of "program", or a set of instructions written in some interactive language; and,
- d) the *sub-schema* - Σ . The function of the *sub-schema* is to generate an *abstract image* of the *data base* which contains only those parts of the *data base* which are directly relevant to the user U, and in a form most suitable for him. The image of D formed by a certain sub-schema Σ will be denoted by D/Σ .

The need to present the user with an image of D, rather than D itself, was explained by the DBTG report [1], which also introduced the concept of sub-schema. However, the DBTG sub-schema can form only "partial-image" of the DB, whereas we claim here that an "abstract image" of the data base is necessary, a term which is explained in [2]. Some ramifications of this requirement will be discussed later.

The process of interaction between the data base D, and a user who "sees" an image D/Σ of it, can be viewed as a sequence of operations, on D/Σ , which are invoked by the user's program Π . We will later see that this is too simplistic a view, but first we will try to understand what is an "operation on the data base".

2.3 The primitives of U-D interaction

In this section we will discuss the operations which should be available to users for their interaction with a DB. Conventionally, these are essentially *update* and *retrieval* operations. Although some data base systems feature very sophisticated techniques by which one can select and identify parts of the DB, notably Codd's "relational calculus" [2], the action performed on these parts is practically always either update or retrieval.

While this *update-and-retrieval* oriented approach is natural for files, it is not suitable for data bases, as the following example might illustrate.

Let $P = \{p\}$ be a set of people and $J = \{j\}$ a set of jobs in some corporate DB. Consider a user U who has the authority to appoint people to jobs. How does he, or his secretary, record such an appointment in the DB? The problem is that in order for an appointment to be recorded consistently, many parts of the DB might have to be updated. Some of these parts might not be known to the user, but even if all the relevant details exist in the user's view of the DB, we would probably not rely on him to carry out the transaction correctly, not to mention that our user would prefer not to go through all this trouble. The solution to this problem is obvious, and

well-known: to provide the user with a procedure (or operator), $appoint(p,j)$, which does all the necessary work involved in appointing p to job j . But under most DB systems such a procedure seems to be viewed as a kind of *utility*, not being an organic part of the DB itself, and which the user can use if he wishes, but does not have to. Our point here is that this kind of procedure must be considered as an integral part of the DB. Moreover it is an essential part of the *abstract view* that the user should have of

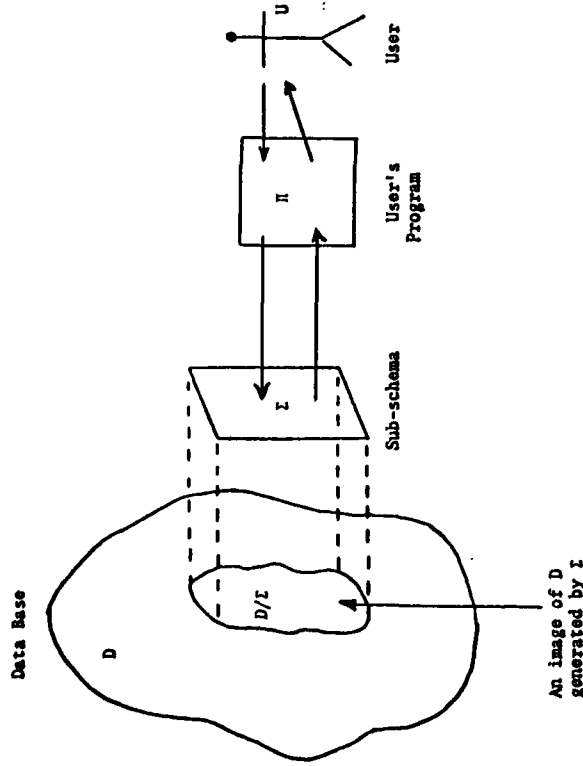


Figure 2.1: A schematic model for U-D interaction

the DB. It is well-known that in order to form *abstractions* in programming it is not enough to create complex data structures, one must provide the operators which manipulate these structures, (see Liskov and Zilles [4], for example). In our example, in particular, the procedure *appoint* should be viewed as one of the operators which can be applied to a pair of abstract objects j4j and pcP. The failure of the DBTG report to supply the user with operators which are suitable to his view of the data base may be the most serious flaw in their proposal. Indeed the sub-schema of DBTG report can only generate a partial image of the DB, not an abstract one, (cf. [2]).

It should be pointed out, however, that the need to present users with an abstract view of the DB is not being completely ignored. This need is usually supposed to be satisfied by classifying the community of users into groups, each of which needs a different level of abstraction. For example, a popular classification distinguishes between *programmers* and *non-programmers*: The programmers view of the DB through a sub-schema and are supposed to manipulate it by direct update and retrieval, (the DBTG report is oriented towards this class of users); while non-programmers, such as airline-reservation clerks, may be several levels removed from the DB. They are supposed to interact with a program, written by a *programmer-user*, which in effect generates the necessary abstraction for the non-programmer. While this approach has its merits, it does not solve our problems. Because even the *programmer-user*, who may have a detailed view of one part of the DB, might need a higher-level abstraction of another part. ⁽¹⁾ We need therefore the ability to generate abstractions of different parts of the DB, on different levels. It is primarily for this reason that the naive update-retrieval approach to U-D interaction is replaced here by the following approach.

(1) Moreover, even a module of the DB itself might need an abstract view of other parts of the DB.

The primitives of U-D interaction are operators, to be called *D-operators*, which are supplied to the user by the sub-schema. In general, a D-operator may have two types of effects: it may affect the DB itself, which is a generalization of the conventional "update", and it may transmit information into the storage space of the user's program, which will be called the *outcome* of the D-operator, and which is a generalization of "retrieval". (It is very important to realize that the D-operators are as much part of the DB as the information on which they operate).

2.4 "Access-control" as protection mechanism

The conventional approach to protection of data bases is based on the following assumption: *If the data base itself is well structured, then all we have to do to protect it is to guarantee that only "legal operations" are performed on it.* According to this assumption, the protection mechanism can act essentially as a mediator between the DB and the user's program Π , monitoring all the D-operations invoked by Π ; thus the name *access-control* for this mechanism. This mechanism must have the authority to reject any operation applied to D, but it has no say as to what happens within Π itself. Thus, in principle, the user's program can be viewed as a "black box" by such protection mechanism, since only the D-operations issued by it seem to be relevant.

Unfortunately, in spite of the intuitive appeal of this approach it turns out not to be satisfactory. We will show in the following section that access-control has some inherent limitations, and cannot be used as a universal protection mechanism. In fact, we will show that even the basic assumption mentioned above is not completely justified, in spite of its apparent obviousness. But first, we will examine more closely the

access control mechanism itself.

The sub-schema I can be viewed essentially as a set of "capabilities", whereas by "capabilities" we mean, loosely speaking, a set of D-operators and rules about their permissible arguments. Since different users may be entitled to different capabilities, the first step of every session of U-D interaction must be the coupling of the user with a certain sub-schema I , to which he is entitled. (1) Once such coupling is done, all D-operations issued by Π would be validated against the capabilities in I .

Following the observation made by Conway et. al. [5], that many D-operations can be validated statically, thus increasing the efficiency of protection; the basic access control mechanism is sometimes refined by performing some checking at "compile time". This method can be viewed as a *compile-time-access-control*. Only the instructions which would become D-operation at run time are examined, the rest of the user's program is still viewed as a black box by such protection mechanism.

(1) Assuming that the identity of U is known, the security of this step is not problematic.

2.5 Limitations of the access-control mechanism

The limitation of "access-control" as a protection mechanism are due primarily to the fact that the process of U-D interaction cannot, in general, be viewed just as a sequence of independent invocations of D-operators. What the user can or cannot do to the data base, or get from it, depends on such things as how is the outcome of one D-operator used by another, and how is it processed by the user's program itself. Such things are beyond the reach of the access control mechanism. To be more specific, we will present below several classes of difficulties that one has with the access control protection mechanism. Some of these difficulties can be overcome within the realm of access control, but with significant cost; others cannot.

2.5.1: The loss of "implicit information"

Consider a D-operator F , and let v be a variable in the storage space of Π . Suppose that F is invoked by Π and that the outcome of F is stored in v . We claim that the variable v carries two types of information if it: First there is the sequence of bits actually stored in v , which we will call the *explicit information* in v . In addition, v carries an important *implicit information*, which is the fact that the content of v was generated by the D-function F ; in a sense, this *implicit information* provides the interpretation for the *explicit information* stored in v . For example, if F is a retrieval operator which retrieves a name from a set of patients names in a medical data base, then we know that v contains a name-of-a-patient. Without this *implicit information* the content of v might be interpreted in any number of different ways. Normally we accept such implicit information as self evident, and we use it without giving it any thought. But as so many other things, the *implicit-information* must be noticed when it is lost, which

may happen as follows:

Let G be another D-operator in our data base, which admits one parameter. Suppose that G is invoked by Π , with v as a parameter. The invocation $G(v)$ obviously communicates to the operator G the *explicit information* in v ; but since G has no way of knowing how v was generated, the *implicit information* in v is effectively lost for G .

To see the significance of such loss of implicit information suppose that F is a retrieval operator which retrieves a member of a set E stored in D . Suppose also that the D-operator G is allowed to operate on members of E only.⁽¹⁾ Now, the program which invokes $G(v)$, after storing an outcome of F in v , "knows" that v does in fact contain a member of E ; and, therefore, that G can be legally applied to it. But the data base, or the operator G if you will, does not share this knowledge, since Π is able to store anything it wants in v . Therefore, to guarantee its proper operation, G must check its argument for membership in E , thus regaining information which the user, or his program, already have, (see figure 2.2)

Essentially, the reason that this apparently wasted effort is necessary, is the freedom that the user's program has in dealing with its own information space. Were it possible, for example, to impose on Π the restriction that nothing but an outcome of F can be stored in v , it would not be necessary for G to check the content of v . (Note also that no implicit information is lost if G is applied directly to F , namely by $G(F)$, which does not give to Π any opportunity to temper with the outcome of F before it is used as an argument of G . But one cannot restrict the user to only this kind of use of the D-operators). This suggests that the implicit information

(1) This is a somewhat abstract example of a type of privacy and integrity rules that one might have, more specific examples will be given later.

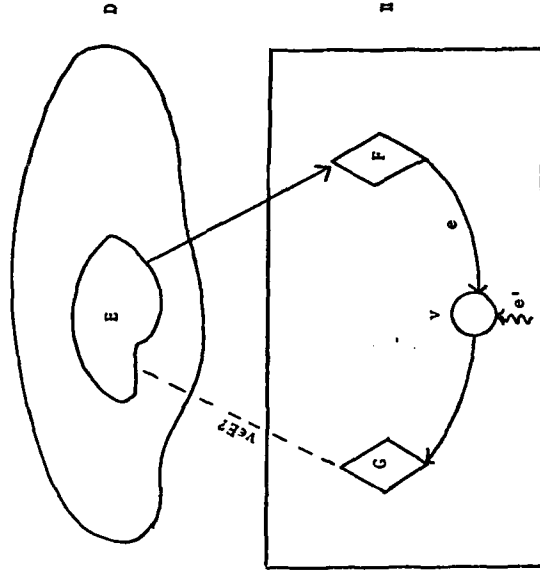


Figure 2.2: A loss of "implicit information"

(In this illustration, diamond shapes represent D-operators which are accessible to Π , and circles represent variables in the storage space of Π .)

Here, the outcome e of the operator F , known to be a member of E , is stored in v . v is then used as an argument of G . G which is allowed to operate on members of E only, must check explicitly that $v \in E$ (broken arrow) because it cannot be sure that its argument is not some e' (wavy arrow) not generated by F .

contained in data retrieved from the data base might be preserved by imposing restrictions on the manner in which Π manipulates this data. Later we will present a technique for doing that, but since it is clear that any such restriction will have its cost, it is desirable to have a quantitative measure for the "value" of the implicit information that such a restriction is supposed to preserve.

We define the value of an implicit information to be the effort that it takes to regain this information, if it is lost, multiplied by the number of times that this effort must be exercised. For instance, in our example above, the value of the information that " v is a member of E ", is equal to the effort that it takes to verify whether a given object e is a member of E . This in turn depends on the manner in which E is defined and stored, and it may be substantial. Note, for example that it is sometimes much more expensive to check if $e \in E$, then to select a member of E . Thus the value of an implicit information may be much higher, than the effort to get it initially. In addition if the same variable v is used by another D-operator, which has the same restriction on its argument, then the value of the implicit information in v is doubled. We will see below some specific applications for the concept of the value-of-implicit-information.

To summarize we can formulate the following principle: The "implicit information" carried by the outcome of a D-operator into the storage space of Π cannot be carried back into the data base without imposing restrictions on the manner in which Π manipulates his own storage space. The cost of this loss can be measured by the "value" of the implicit information.

2.5.2. The problem of value-dependent rules

One of the most serious difficulties in the protection of data bases is the fact that many of the integrity and privacy rules which one would like to impose are "value-dependent". The enforcement of such rules is very expensive, first because it requires run-time checking, and secondly because such checking may be involved with an extensive data base search. It was already pointed out by Conway, et. al. [5] that one can increase the efficiency of protection by distinguishing between *value-independent* and *value dependent* rules, resolving the former as early as possible, say at compile time. But there is a lot more which can be done. We will show below that if the *implicit information* of the outcomes of D-operators can be preserved, then it is sometimes possible to turn apparently *value dependent* rules into effectively *value-independent* ones, which can therefore be enforced much more efficiently. We will clarify this point by an example.

Consider a medical data base D , and a set PAT of patient names in it. Let $A_1 \dots A_n$ be the attributes of a patient $p \in PAT$. Since these attributes do not have to be stored together nor retrieved together, we will consider A_i to be D-functions⁽¹⁾ defined over PAT . Now, let the user U be a doctor, and let $PAT(U)$ be a subset of PAT which consists of the names of all patients of doctor U . Suppose that the interaction of U with D is subject to the following privacy rule, to be called r .

$r = "U$ is allowed to retrieve the names and attributes of his own patients only".

(1) Whenever appropriate we will replace the term "D-operator" by "D-function", with the same meaning implied.

From the point of view of access-control protection mechanism this rule is equivalent to the following $n+1$ value dependent rules.

$r_0 = "U$ can get a p -PAT only if $p \in \text{PAT}(U)"$, and
(for $i=1 \dots n$) $r_i = "U$ can invoke $A_i(p)$ only for $p \in \text{PAT}(U)"$.

The rule r_0 can be enforced, for example, by providing U with a D-operator, MYPAT, which acts as a filter, communicating to $\Pi(U)$ only names which belong to $\text{PAT}(U)$. Note that the enforcement of r_0 does not automatically enforce r_i . U can still get a patient's name p' from an outside source and he can apply A_i to it. Thus whenever $A_i(p)$ is invoked, for any p , a check must be made whether or not $p \in \text{PAT}(U)$. Thus r_i are indeed value-dependent rules, which must be resolved at run-time. Suppose, however, that it is possible to impose a restriction on Π so that a certain set of variables in its storage space could contain only an outcome of the operator MYPAT. If v is such a variable, then it is known that v contains a member of $\text{PAT}(U)$. Therefore if $A_i(v)$ is invoked, no further checking of v is necessary. (The implicit information in v is not lost in this case, since by our assumption Π cannot store in v anything but an outcome of MYPAT. All we need now, in order to enforce r_i , is to guarantee that the argument of A_i is one of these monitored variables, which as we will yet see, can be done at compile time. Thus the rules r_i become effectively value-independent rules. (1) This result is achieved essentially by preserving the implicit information which is due to the rule r_0 . (We will show later how this can be done in practice).

2.5.3: "Intentional resolution" of privacy protection

The previous discussion was relevant to both privacy and integrity of data bases, in this sub-section we will be concerned exclusively with privacy.

(1) Note that the saving here can be measured again by the "value" of the implicit information of a p retrieved from $\text{PAT}(U)$.

One of the most important characteristics of privacy protection mechanisms of data base systems is the fineness of the privacy measures specification, which may be called the resolution power of the privacy protection. This aspect of privacy protection has two independent dimensions. One dimension, which may be called structural resolution is related to the flexibility of the specification as to *which parts of the data base should a given user be allowed to get*. A good privacy protection mechanism should provide the user with every bit of information that he is *entitled* to get, regardless of how this information is distributed in the data base; while hiding from him everything else. The other dimension of the resolution power of privacy protection is related to *what is the user allowed to do with a piece of information supplied to him by the data base*. The ability to condition the supply of data on its intended use, and the ability to guarantee that this intention is not violated, will be called intentional resolution. It is this aspect of privacy protection which will concern us here.

It is obvious that once a piece of information is revealed to the user himself, there is no way for the data base to impose restrictions on how it is used. The intentional-resolution for such information becomes a legal and moral matter. But much of the information retrieved from the data base is not intended for the user's eyes, but for his program. It frequently happens that confidential information is needed in the user's program as an accessory for getting other, sometimes less confidential data, which is what the user is really after. Were it possible to prevent the user's program from revealing to the user himself certain information given to it by the data base, one would be able to release to the program

data which for privacy reasons should not be released to the programmer himself. In the absence of such *intentional resolution* capability one is forced to release either too much or too little information to the requesting program. An extreme, but very common example of the former case is the almost universal permission one is forced to grant system programmers to retrieve any information stored in the data base. This is necessary because it is the job of these programmers to perform various routine manipulations of the data under their control. Thus, although most of these programmers have no need to actually see the information manipulated by them, they can usually do so, which according to many observers, presents the most serious threat to the privacy of data bases. We would like, therefore, to be able to prevent a programmer from seeing certain information which is released to his program. In other words, we sometimes want the programmer to "eat the cake but not to have it in his stomach." Which is one aspect of *intentional resolution*.

Unfortunately, the access control protection mechanism is fundamentally incompatible with intentional resolution, because it does not distinguish between the human user and his program. As was already pointed out the access-control mechanism has the authority to decide which information could be retrieved from the data base, but it has no say about how such information is to be used, either by the human user or by his program. Indeed the access control mechanism has no more control over the user's program than it has over the user himself.

Therefore, it is clear that in order to achieve any degree of *intentional resolution* we must let the data-base have some control over what happens inside the program which interacts with it, restricting the program as to what it can do with information retrieved from the data base.

Thus, we are arriving at the same conclusion from a fundamentally different point of view, namely that some control must be imposed by the data base on a program which interacts with it. Later we will consider some examples which show how such a control can indeed provide intentional resolution. For more detailed study of this particular subject see [6].

2.6: Conclusion

It has been shown that the access-control protection mechanism is not satisfactory. It makes the enforcement of certain types of data base rules very expensive, due to loss of "*implicit information*"; it leaves us with unnecessarily large numbers of *value-dependent* rules, which must be enforced at run time; and it does not provide at all, for *intentional resolution* of privacy protection. The alternative which almost suggests itself is to impose some restrictions on the programs which interact with the data base, and not just on the D-operators which they invoke. This, however, would require a fundamentally new approach to the process of U-D interaction. Such an approach will be developed next.

3: Another approach to U-D interaction

3.1: An outline of the proposed approach

Since we are looking for a way to impose restrictions on the internal behavior of the user's program, it is only natural to expect some assistance from the language in which this program is written. This is a sensible expectation because languages have ways for imposing a discipline on the processes of computations generated under them. This is done in a variety of ways: by "types", by "scope-rules" etc. For example, no ALGOL program can cause a "real" number to be stored in an "integer" variable, (provided that the rules of ALGOL are strictly enforced). Indeed languages can be characterized in terms of such discipline: On a certain level of abstraction, which ignores syntax as well as many semantic issues, a language can be defined in terms of the *capabilities* (1) that it provides to its processes of computations. By this we mean, loosely speaking, that the language can be viewed as a system which has a collection of primitive data objects and operators built into it, together with rules which specify which operators can be applied to which objects and under which circumstances.

This observation raises another problem: A given language L would usually have no intrinsic capabilities which allow interaction with a specific image D/I of D , such capabilities presumably exist only in Σ . How, then, can a program written in L , interact with D/I ? To answer this, rather formalistic question we now present a schematic model for U-D interaction, which turns out to be very convenient for the solution of the other problems at hand.

(1) The term "capability" is used here not in its strict technical sense of Lampson [11] say, although it is not far removed from it.

Let L be a programming language, to be called the "*base language*", whose role in our model would be somewhat analogous to the role of the "host language" in the conventional approach. A user who wishes to interact with the image D/I of D , using the language L for that, must first perform a coupling between L and Σ . The act of "coupling" will be viewed as an *amplification of the language L* by augmenting it with the capabilities of Σ , thus creating a new "r-amplified" language, to be denoted by L/Σ . The newly created language, which would exist only for the duration of a single session of interaction between U and D , would have the same syntax and most of the semantics of the base language L . But it is truly a new language since it contains capabilities which do not exist in L . The user can now write a program in L/Σ , which will be able to interact with D/I , due to the capabilities achieved from Σ . (Figure 3.1 is an attempt to illustrate this model). In this way, the user program would be under the joint jurisdiction of L and Σ ; which, in principle, at least is what we are after.

To carry out this general plan we must first have concrete models for the intrinsic capabilities of the language L , and of Σ . Such models will be developed in the following two sub-sections.

3.2: The "capabilities" of programming languages.

Let L be a language, and let Π be a process of computation (1) generated by a program written in L . If we take a snapshot of Π at a certain stage of its execution, we can talk about the *storage space* of Π , to be denoted by S , (or $S(\Pi)$), which is the set of all *information objects* (or simply *objects*), which are accessible to Π at this moment, (the concept "information object" will be left undefined here). $S(\Pi)$ can be manipulated by means of the *primitive operators* provided by L to every (1) Although we sometimes call Π "a program" we really mean a "process of computation".

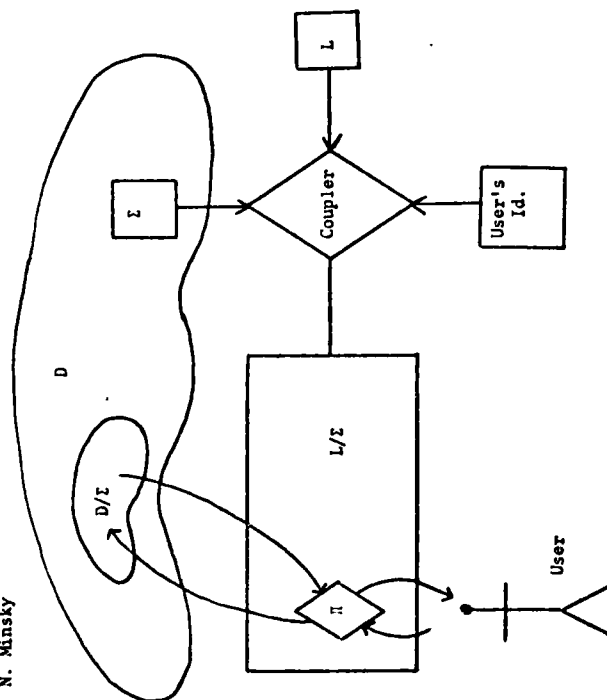


Figure 3.1: A schematic model for the proposed approach to U-D interaction

(Diamond shaped figures represent active components). The coupler amplifies the language L by the capabilities in Σ , generating the new language L/Σ . The user program Π operates within L/Σ , so to speak. It has, therefore, the necessary capabilities for interaction with D/Σ .

$\Pi(L)$. The set of all such operators will be denoted by P_L . Examples of such operators are: the "+" operator which adds two numbers, generating a new object (number) which contains their sum; or a *declare* operator which generates new variables in $PL/I(1)$. (Note that we are talking about the operators in the level of the language L , not about lower level operators which might be used to implement L).

It would be convenient to consider some of the objects in S as being *primitive objects* provided by L to every $\Pi(L)$, such as the various literals recognized by the language. This set of objects will be denoted by S_L .

Ignoring transfer of control, which does not directly change any object in S , the process Π can be formally described as a sequence of operations of the form:

$$s_g \leftarrow p(s_1 \dots s_n) \quad (n \geq 0)$$

which is the application of the operator $p \in P_L$ to the objects $s_1 \dots s_n$ in S . Such an operation may affect the objects s_i ; in addition it generates a new object s_g , to be called the *outcome* of p , and which may in particular be null. (We are making here the simplifying assumption that an operation can generate at most one object).

3.2.1: Application rules: Under most languages the process Π is not free to apply an operator to any set of objects. The language usually has a set of rules built into it which specify which operators can be applied to which objects, and under which circumstances. This set of rules, to be called the *application rules* of the language, will be denoted by R_L . Languages may vary widely by the type and form of their application rules, and many important properties of a language may be modeled by these rules. Here we will discuss only a certain type of such rules.

(1) At this point it is irrelevant for us whether an operator can work only at "compile time", see however below.

But before we go into specifics it should be pointed out that most languages would have different sets of rules for the two major phases of the life span of a process: the "compilation" phase and the "running" phase. Operations which are allowed during the first phase, such as storage allocation, might not be allowed during the second phase, and vice versa. Except of this variability of the application rules there is no reason for us to distinguish between these two phases.

To demonstrate the significance of the application rules let us consider an example of a language which has practically none — a "machine language."

Let L be the machine language of a classical Von-Neumann machine. The storage spaces of every $\Pi(L)$ is fixed in this case, being the set of memory cells of the machine. The set of operators, P_L available to Π are the machine instructions. There are practically no application rules here, Π is free to apply any operator to any cell. This, however, does not mean a complete anarchy, since the operators themselves may have their own rules built into them. For example, a division operator would usually refuse to divide by zero. Such *internal* rules will not be discussed in this paper, we are going to consider the various objects and operators as black boxes, considering only application rules which are imposed by the language from the outside, so to speak.

3.2.2: Brands: Rules are usually most effective when they are general, namely when a single rule is applicable to a large number of cases. To facilitate such rules it is useful to introduce an explicit classification of all objects in S to a set of equivalent classes so that the rules will be defined in terms of these classes. We will do that, by associating a

special brand, β , with every object in S . The set of all such brands in the language will be called B_L^* . Every object seS would thus have two parts (1): $brand(s)$ which is some $\beta \in B_L^*$, and $value(s)$ which is the body of the object. An object in S whose brand is β will be called β -branded object, or just β -object. Although the purpose of this branding is to divide S into large classes, it is sometimes convenient to consider an object as a class of its own. For simplicity we will use the name of such an object as its brand, (e.g. "+" would be considered as the name as well as the brand of the "addition operator") All other brands in B_L^* will be denoted by barred symbols, e.g. \bar{i} -for integer, \bar{r} -for real \bar{f} -for text, etc. The set of all such brands will be denoted by B_L , which is therefore a subset of B_L^* . We will assume that a brand $\beta \in B_L$ is attached to an object when the latter is generated by Π , and that this association cannot be changed afterwards. In addition, we assume that the brands themselves do not admit any operation, except that they can be compared for equality.

All that can be summarized as follows: On some level of abstraction, a language L can be characterized by the *capabilities* that it provides to its processes of computation. "Capabilities" are defined as the 4-tuple.

$$(S_L, P_L, B_L, R_L),$$

where, S_L — is the set of primitive objects in L ;

P_L — is the set of primitive operators;

B_L — is the set of brands;

R_L — is the set of rules of the language

(1) Note that this is just a formal device. In practice, as we will see later, one can frequently avoid the use of an explicit brand.

In addition, there is the set B_L^* , which includes B_L and a set of distinguished symbols from $S_L \cup P_L$ which are considered to be brands.

We now proceed to examine an important class of very simple but useful rules.

3.2.3: "Simple" Application-rules: What we will call "simple rule".

has the following general form:

$$r: \beta_g \leftarrow (\beta_0, \beta_1 \dots \beta_n) \quad (\text{for } n \geq 0)$$

where $\beta_i \in B_L^*$, for $i=0, 1 \dots n$; and $\beta_g \in B_L$.

The rule r (r is a label, used for discussion only) serves two functions:

First, the right hand side of r serves as a right for $\Pi(L)$ to invoke any operation $p(s_1 \dots s_n)$, such that brand $(p) = \beta_0$ and brand(s_i) = β_i , for $i=1 \dots n$. The right hand side of r will accordingly be called right and will be denoted by ρ ; and it is assumed that no ρ appears more than once in B_L . The second function of the rule is to determine the brand of the outcome of an operation $p(s_1 \dots s_n)$, to be the β_g of the rule which contains the right to it, (by the assumption above, there is at most one such rule). If β_g is null it would mean that no new object is generated into $S(\Pi)$, even if p does have an outcome.

These rules are simple, and restricted, in several respects, in particular:

a) The rules are invariant of the values of their arguments; b) the rules are variant of the state of computation, in particular: they do not dis-

tinguish between the "compilation phase" and the "running phase" of Π ;

c) the rules have "global validity", thus they cannot be used to model

the "local" nature of "looping". Yet, the "simple rules" can model a significant part of

the production rules of many languages. For example, if \bar{I}, \bar{R} and \bar{C} stand

for the brands of integer, real and complex objects, then the following are

samples of the rules which we might have

$$r1: \bar{I} \leftarrow (+, \bar{I}, \bar{I})$$

$$r2: \bar{R} \leftarrow (+, \bar{I}, \bar{R})$$

$$r3: \bar{C} \leftarrow (+, \bar{C}, \bar{C})$$

$$r4: \bar{R} \leftarrow (IM, \bar{C})$$

The rule $r1$ allows Π to add two integers, specifying that the outcome is an integer. $r2$ allows Π to add an integer to a real number, (obviously there is a conversion to be done here, but this is irrelevant to the application rules). $r4$ allows Π to apply the operator "IM" to a complex number, which presumably returns its imaginary part. Note also that if the set R_L at hand does not contain the right $(+, \bar{I}, \bar{C})$, then Π cannot add an integer to a complex number.

In order to get any approximation to realistic programming languages we must introduce explicitly the concepts of *variable* and *assignment operator*. This we do next.

3.2.4 Variables, and the assignment operator: The concept of variable

can be introduced into our model, intuitively, as follows: A *variable*

is an object whose value is another object. The *value* of a variable is

assigned to it by the *assignment operator* "=". In our notation the

assignment operation would, have the form $:= (\text{variable}, \text{value})$, (for

simplicity we assume that this operation has no outcome). If a variable

appears as an argument of an operation in any other context, then the

object which is its value is assumed. An important property of a variable

is its *scope*, which is the set of all objects which can possibly serve

as values of the variable. The scope of a variable ⁽¹⁾ is clearly determined

(1) At least an "upper bound" of the scope is determined by the application rules. Note that the operators themselves may have further restrictions not reflected in the application rules.

by the rules in R_L , as the following example shows:

Let \overline{IV} be a brand in B_L , and suppose that the following rule exists in R_L :

$$r: (:=, \overline{IV}, \overline{I})$$

This rule allows \overline{IV} -objects to appear "at the left hand side of an assignment statement"; which makes them, at least from the application-rules point of view, into variables. In addition, the objects assigned to such a variable, according to r , must be \overline{I} -objects. If moreover there is no other rule in R_L such that the first two components in it's ρ are ":=:" and " \overline{IV} ", then no other type of object can possibly be assigned to an \overline{IV} -object. Thus the scope of any \overline{IV} branded variable is the set of all \overline{I} branded objects.

In general, if the scope of a variable is the set of all β -objects, for a given $\beta \in B_L$, we will call it a β -variable. We will also say that such a variable is a *simple variable*. It is clear that if an object s is stored in a simple variable, then its brand does not have to be explicitly stored. Moreover if all the variables in π are simple, then "simple application rules" can be enforced at "compile time".

These properties are the main reasons why in many programming languages all variables are simple, e.g. *integer variables*, *real variables*, etc. In fact, simple variables are so common that one tends to forget the difference between the *type*(brand) of an object and a *scope* of a variable; indeed the term "type" is usually used for both, although it is clear that an "integer number", for example, and an "integer variable" are not similar objects and do not satisfy the same rules. The difference between the two concepts surfaces when one deals with non-simple variables. A good example is the confusion associated with the "mode union" in ALGOL-68, which is not a "mode" in the same way that "mode integer" is. No such confusion would

exist in our framework, as the following example shows: Let \overline{IRV} be a brand in B_L , and suppose that R_L contains the following two rules.

$$r1: (:=, \overline{IRV}, \overline{I})$$

$$r2: (:=, \overline{IRV}, \overline{R})$$

If there is no other rule which allows an assignment to \overline{IRV} objects, then the scope of these objects, as variables, is the union of integer and real numbers. Thus, there is no formal difference between simple and non-simple variables. What distinguishes between them are the rules.

In conclusion, it should be pointed out, again, that "simple application rules" are not sufficient even to model some of the most basic rules built into conventional programming languages. Our contention is, however, that these rules can serve as a convenient starting point from which one can generalize in various directions. In particular, one needs *value dependent rules*, which depend on the values of the arguments of an operator, not just on their brands. Another vital generalization would lead to *context dependent rules*; rules which are valid only within certain modules of a program, or during certain phases of the lifetime of a process, (an example of such generalization will be discussed in section 4). Other types of generalizations are also possible, (see section 5). But due to space limitations we will confine ourselves in this paper mostly to "simple rules."

3.3 A model for the sub-schema, and the generation of L/Σ

The sub-schema will be viewed here essentially as a collection of capabilities which facilitate interaction with a given image of the data base, and which are designed to augment the capabilities of a given language $L(1)$. Formally, Σ is defined to be the 4-tuple (S_L, P_L, B_L, R_L) , where:

S_L — is a set of (names of) information objects in D , which are to be accessible to the user. They will be called *D-objects* (e.g. sets, relations, records of various types, etc.).

P_L — is the set of D-operators.

B_L — is the set of brands, which are all different (2) from the brands in B_L . (For convenience we define the set $B_L^* = \bigcup_{D \in \Sigma} S_D \cup P_D \cup B_D$, similarly to B_L^*).

R_L — is a set of rules, to be called D-rules, designed to regulate the interaction of Π with the data base. Initially, we will restrict ourselves to "simple rules", which have the following form:

$$r: \beta_g \leftarrow (\beta_0 \dots \beta_n)$$

where $\beta_i \in B_L^* \cup B_L^*$ for $0 \leq i \leq n$, and

$$\beta_g \in B_L \cup B_L^*.$$

This is exactly the structure of the "simple-rules" in R_L except that these

(1) This does not mean that the interaction with the database must be carried out by means of a single language, different sub-schemas can be designed for different languages.

(2) To distinguish between the brands in Σ from those in L , we will denote the former by barred lower case letters, e.g. \bar{a}, \bar{b} ; while the latter are denoted by upper case barred symbols. When we do not wish to distinguish between B_L and B_L^* we will use $\beta, \beta_1, \beta_2 \dots$ to denote brands.

rules can use the brands in Σ as well as those in L .

This concludes the formal definition of the sub-schema. The coupling with Σ can now be viewed as an augmentation of the capabilities of L , namely the sets S_L, B_L, P_L, R_L , with the corresponding sets S_L, B_L, P_L, R_L . This augmentation creates the Σ -amplified language L/Σ in which the user's program $\Pi(1)$ is written. Such a program has an access to a richer set of primitive data objects, $S_L \cup S_L^*$, then does a program written in the original language L ; and a richer repertoire of primitive operators, $P_L \cup P_L^*$, which allows it to manipulate the data base. It is also assumed that there is a mechanism which guarantees that Π executes only operations which are authorized either by R_L or by R_L^* . Thus, Π is under the joint jurisdiction of the language L and the sub-schema Σ .

One of the main reasons for introducing the above model was to give the data base control over what happens with information transmitted to the user's program. Here is how this can be done:

Let T be a D-operator, and let

$$r: \beta = (T, \dots)$$

be a rule in R_L . Let $t \leftarrow T(\dots)$ be an operation authorized by r . The object t generated by T (which is simply the information retrieved by T from the data base), would, by rule r , be branded by β . To see what the future of t might be, we will distinguish between two cases, depending on β .

First, if β is a member of B_L then there would be rules in R_L which specify how such objects can be manipulated. For example, if β is \bar{a} then this object can be manipulated as any other real number. In general, an outcome of a D-operation which is branded by one of the standard brands of B_L is released from the control of the sub schema and gets to be under the

(1) From now on, Π will denote a program written in L/Σ , or rather, the process of computation generated by such a program.

jurisdiction of the language L.

Suppose now that β is a member of B_L . In this case there is no rule in R_L which authorizes any operation on t . The only operations which are applicable to t are those which are authorized by rules in R_L . Thus, although the object t resides in the storage space of the user's program, it is still under the jurisdiction of the data base. For example, the program Π would not be able to multiply, compare or print the object t , unless such operations are permitted by R_L .

Before considering an example, we will introduce some conventions and notations which will simplify the subsequent discussion.

a) For every brand β , explicitly included in B_L , we assume that there is a companion brand in B_L , say β^v , which is the brand of β -variables. The necessary rules and operators which make β^v -objects into β -variables are assumed to be included in L . Thus a program Π written in L/L should be able to generate (declare ?) β -variables, for every brand β explicitly included in B_L ; much in the same way as "integer-variables" are declared in ALGOL.

b) It is sometimes necessary to generate a copy of the value of an object, assigning to it another brand. We introduce a special operator for that, called COPY.

The rule:

$$\beta^2 \leftarrow (\text{COPY}, \beta^1)$$

would mean that COPY can be applied to any β^1 -object, generating a new object s^1 such that $\text{brand}(s^1) = \beta^2$ and $\text{value}(s^1) = \text{value}(s)$. The operator COPY is assumed to be included in every L .

c) Let E be a set of object in D , and suppose that we want to specify that every member of ecE must be branded by β when copied to the storage space of Π . There may be several operators which can retrieve members of E , but sometimes it would be irrelevant for us which operator is actually used. For such cases we introduce the operator GET to be a "universal

retrieval operator" from a set. For example, the rule

$$\beta \leftarrow (\text{GET}, E)$$

would mean that any member retrieved from E would be β -branded. Similarly, we introduce a "universal storage operator", PUT. For example, the rule

$$\leftarrow (\text{PUT}, \beta, E)$$

would mean that any β -object can be stored into the set E . (Note that PUT does not have an outcome). The operators GET and PUT will be used here primarily as notational devices.

d) To this point we did not introduce any operator on the brands themselves. In particular, there is no explicit way for Π to compare two brands. There is one exception, however, every brand can be compared with a special null brand. In addition, every operator can produce a null-branded object, which will usually be used to signal an end of list or set.

To clarify all that we will now consider an example:

Let T be a binary tree, stored in D , which consists of the set of nodes $\{n\}$. Let LLINK, RLINK and DATA be D-functions defined over the nodes $n \in T$, such that the outcomes (values) of LLINK(n) and RLINK(n) are the left and right sons of n , respectively; and the outcome of DATA(n) is the data stored in node n , which for simplicity is assumed to be an integer number. (n itself may be viewed as a direct pointer to a node, so that the function DATA, say, may be applied to any $n \in T$, not just to some "current node"). Let tree(n) be a subtree of T whose root is the node n . The following subschema enables the user coupled to it, to retrieve the data(n) of all nodes $n \in \text{tree}(N)$, for a given node N . Moreover, as we will see, only tree (N) would be accessible to the user.

$$Z = \{S_L, P_L, B_L, R_L\}$$

$$S_L = \{N\}$$

$$P_L = \{LSON, RSON, DATA\}$$

$$B_L = \{\bar{n}\}$$

$$R = \{r1: \bar{n} \leftarrow (COPY, N)$$

$$r2: \bar{n} \leftarrow (LLINK, \bar{n})$$

$$r3: \bar{n} \leftarrow (RLINK, \bar{n})$$

$$r4: \bar{I} \leftarrow (DATA, \bar{n})\}$$

First, note that there is a brand \bar{n} defined in Z . A program Π written in L/Z can therefore generate \bar{n} -objects. By convention (a) above Π can also generate (or declare) \bar{n} -variables, (which are variables whose scope consists of \bar{n} -objects). Let us see now how \bar{n} -objects can be generated, and how they can be used.

First, the node N itself can be \bar{n} -branded (by rule r1). The only other way to generate \bar{n} -objects is by means of the D-operators $LLINK$ and $RLINK$, (by rules r2, r3). \bar{n} -objects can be used only as parameters for the three D-operators $LLINK, RLINK$ and $DATA$. The outcome of $DATA$, in particular, is an \bar{I} -object (namely a normal integer), which is therefore released to Π .

It is clear that Π/Z is able to get the data of every node in $tree(N)$, moreover there is no way for it to get the data of any other node. The reader may argue that the same result can be achieved as easily by traditional access-control, as follows: If the only D-operations available to the user are $LLINK, RLINK$, and $DATA$; and if he begins with the node N then there seems to be no way for him to get to any node outside of $tree(N)$. However, this argument does not take into account the possibility

that the user would get a node $n' \notin tree(N)$ from an outside source, and then use it as an argument of $DATA$, say. The only way to prevent that, is to check the argument of every one of the three D-operators for membership in $tree(N)$. This is not only a run-time check, it may be a very expensive check to do, since it may be very difficult to verify whether or not a given n belongs to $tree(N)$. In our case, on the other hand, it is enough to guarantee that the three D-operators are applied to \bar{n} -objects, or \bar{n} -variables, which can usually be done at compile time. Since \bar{n} -object is known to belong to $tree(N)$, which is the *implicit information* contained in these objects, no further checking is necessary. Π can still get a node $n' \notin tree(N)$ from an outside source, but it has no way to brand it by \bar{n} , or to store it in an \bar{n} -variable; therefore such foreign information cannot be used as an argument of our D-operators.

4: Case studies

In order to demonstrate that the proposed model can cope with the problems posed in section 2.5 we will consider now several, more or less, realistic examples.

Example 1: As a more specific example of the problem posed in section 2.5.1, consider the following situation. Let E be a set of text strings in D , and let U be a user who is allowed to retrieve members of E . Suppose also that U is allowed to add members of E to another set E' . He can add to E' any member of E that he wants, the only restriction is that U should not add to E' anything but a member of E . This kind of interaction is supported, and enforced by a Z with the following set of rules. (Here, and in the following we will omit the specification of S_L, P_L and B_L , only R_L will be explicitly specified).

r1: $\bar{e} \leftarrow (GET, E)$
 r2: $\bar{f} \leftarrow (COPY, \bar{e})$
 r3: (PUT, \bar{e}, E')

By rule r1, Π can get any member e of E , which will be \bar{e} -branded. Such an e can be examined by Π since it can be converted to $taxr$, (by r2). But an \bar{e} -object itself cannot be manipulated by Π , it is thus known that it carries a member of E , (namely, its implicit information is preserved). By rule r3, an \bar{e} -object can be added to E' . Note that these rules can be enforced at compile time, if the \bar{e} -objects are stored in \bar{e} -variables in Π .

Note that the above set of rules does not allow the user to add to E' a string which he knows to be a member of E , but which was not retrieved from E . This restriction can be removed in a number of ways. For example, there may be a special operator PUT' which adds text strings to E' , and the rule

r4: (PUT', \bar{f}, E') .

The operator PUT' is thus applicable to any text object, but it should check its first argument for membership in E , before adding it to E' .

Example 2: Consider a school data base, and let $E = \{(c, r, t)\}$ be a class-schedule relation, where c stands for a class, r for a room, and t stands for a time period in which the class is scheduled, (all given in integer codes). For obvious reasons the users are not supposed to modify E arbitrarily, instead, the following procedure is established: A user who wants to schedule a certain class invokes an operator F , with argument which describes the requirements of the user. F , which is a pre-designed D-operator, returns a *feasible* assignment $e = (c, r, t)$ as an "advice" to U . (Every call to F would produce another such "advice" until there is none left). The user can collect as many feasible assignments as he can get and he can evaluate and compare them, choosing, say, e_1 as the best. At

this point an operation $A(e_1)$ can be invoked, A being the *assignment* operator which actually adds e_1 to the schedule.

We have here a similar problem to the above. Normally, A would have no way of knowing that its parameter was suggested by F as a feasible solution. Thus to guarantee the integrity of the whole schedule, the feasibility of the argument of A must be explicitly checked. (Note that it may be more difficult to verify whether or not a given e is feasible, than it is to select a feasible solution). Such run-time check is avoided by the following sets of rules, (which are very similar to those in example 1):

r1: $\bar{e} \leftarrow (F, \dots)$
 r2: $\langle \bar{f}, \bar{f} \rangle \leftarrow \langle C, R, T, \bar{e} \rangle$
 r3: (A, \bar{e})
 r2_a: $\bar{f} \leftarrow (C, \bar{e})$
 r2_b: $\bar{f} \leftarrow (R, \bar{e})$
 r2_c: $\bar{f} \leftarrow (T, \bar{e})$

The only thing which requires an explanation here is the notation used for

r2. The rule r2 stands for the following three rules:

Where the operator C is supposed to get the first component of e , etc.

Note also that the arguments of F , in rules r1, are not relevant for us here.

Example 3: We turn now to the case discussed in section 2.5.2, but in order to make it more realistic, we will present it in more specific terms and with some added complications.

As in 2.5.2, let PAT be a set of patients names, and let $DEP, MED, PERS$, DIS be D-operators defined over PAT , such that for every $pePAT$: $DEP(p)$ retrieves the department of p , $MED(p)$ and $PERS(p)$ return its *medical* and *personal* information, respectively; and $DIS(p)$ discharges p from the

hospital. Let U be a doctor, and let PAT' be the set of names of U 's patients. (PAT' was denoted by $PAT(U)$ in section 2.5.2). Now, suppose that while U is allowed to get all information about his own patients, as well as to discharge any of them. He is allowed to get only the medical data and the department of all other patients. In such a way U would be able to make, say, some statistical analysis of the medical characteristics of all the patients of a certain department without being able to get more personal information about patients not treated by him, or to discharge them. This situation is realized by the following rules:

```

r1:  $\bar{p}_1 \leftarrow (GET, PAT')$ 
r2:  $\langle \bar{f}, \bar{f}, \bar{f} \rangle \leftarrow \langle \langle DEP, MED, PERS \rangle, \bar{p}_1 \rangle$ 
r3:  $\bar{f} \leftarrow (COPY, \bar{p}_1)$ 
r4:  $(DIS, \bar{p}_1)$ 
r5:  $\bar{p}_2 \leftarrow (GET, PAT)$ 
r6:  $\langle \bar{f}, \bar{f} \rangle \leftarrow \langle \langle DEP, MED \rangle, \bar{p}_2 \rangle$ 

```

Note that although \bar{p}_1 -objects and \bar{p}_2 -objects have exactly the same structure, both are names, they would be treated very differently by Π/Z . Note also that all these rules with the possible exception of $r1$ are *value-independent* and can be enforced at compile-time. (As to $r1$, it could have been represented in an explicitly value-dependent way but since we did not introduce value-dependent rules in this paper we chose to hide the value dependency of $r1$ behind the assumption that the set PAT' exist explicitly in D).

Example 4: Although the previous example already exhibits some degree of intentional-resolution, we will consider now a more distinct example⁽¹⁾

(1) The example to be discussed here was originally presented in [6] it is repeated here because it exhibits an interesting generalization of our D-rules, and for the sake of completeness.

of this important aspect of privacy protection.

The Problem: Consider a data base D , and a highly confidential set of records, $E(e)$, in it. Let U be a programmer who is commissioned to apply a transformation T , to be programmed by him, to every record $e \in E$, and to store the transformed records back into the data-base, in a set E' . Suppose that due to the confidentiality of E we do not want its content to be revealed to the programmer U himself, or to be leaked in any form or shape to the outside world. (This problem represents the general need to protect the confidentiality of information against the very people who are employed to process and maintain it. As was already pointed out in section 2.5.3, the solution of this kind of problem has a great practical importance).

The difficulty with this case is that since our user has to program the transformation T himself, the confidential records $e \in E$ must be *released* to his program, so that they can be examined and manipulated. This means, however, that the records retrieved from E would not be under the jurisdiction of the D-rules, and we are back again where we started.

One solution to this problem is to *confine* the user's program, in the sense of Lampson [7]. Namely, to block all the "output channels" of Π so that it cannot communicate to the outside world any information about E . Although such a confinement may be possible, as was shown by Lampson, it is usually not a practical solution; because the same program which manipulates the confidential information, may also be doing other things which might require various output channels. Thus, instead of the complete confinement of the program Π as was suggested by Lampson, we would prefer a "local confinement", just of the particular module (or modules) which perform the transformation T . For that we will need a new linguistic structure in L , to be called "confidence-module", to be introduced next.

Let M be a normal module of a program, such as *block* or *procedure*. We will use the term "output channels" of M , to the variety of means that M

has to affect its environment, such as a global variable or a file.

Normally, one does not have much control over which output channels would a given module have. In ALGOL-like languages, for example, every object defined in a given block is potentially an output channel of every inner block. Even more important is the fact that every file accessible to Π is a potential output channel of every module of Π . We now define a *confidence-module* to be a module which has only a specified set, $c_1 \dots c_k$, of output channels. Syntactically it will be denoted by

confidence ($c_1 \dots c_k$)M,

where M is a normal module. (Some comments about the implementation of such modules where made in [6]). We will not be specific as to the way in which the output channels are specified, but an example may indicate some of the possibilities. Let F be a file, V a global variable and b a brand.

The module

confidence (F,V,b) begin ... end

should be able to access the file F, and to write into the variable V and into any b-object which he can access, but it should have no other way to affect its environment.

To utilize the confidence modules for the problem at hand, a generalization of our D-rules will be necessary. As was already pointed out, the scope of *simple* D-rules is the entire program Π . Namely operations which are legalized by a certain D-rule can be carried out anywhere within Π . We will now introduce the ability to restrict the scope of a D-rule:

If r is a D-rule, then the rule

r_1 : $r/\text{confidence } (c_1 \dots c_k)$

is identical to r but it is valid only within confidence modules which do not have any output channels except $c_1 \dots c_k$.

Returning now to our problem, suppose that R_T consists of the following D-rules:

r_1 : $b_1 \leftarrow (\text{GET}, E)$
 r_2 : $\bar{R} \leftarrow (\text{COPY}, \bar{b}_1)/\text{confidence}(b_2)$
 r_3 : $\bar{b}_2 \leftarrow (\text{COPY}, \bar{R})$
 r_4 : $(\text{PUT}, \bar{b}_2, E')$

The program Π written in L/E would be allowed, by rule r_1 , to retrieve records from E, but every such record would be b_1 -branded. The brand b_1 appears only in one more rule, in r_2 , which is applicable only within confidence modules whose only output channel consists of b_2 -branded variables. Outside such confidence modules the records e retrieved from E cannot be manipulated by Π . (see Figure 4.1.) Let M be such a confidence module. By rule r_2 the b_1 -branded objects, namely the records $e \in E$, are *released* within M as normal real numbers. (1) Therefore, within M it is possible to program an arbitrary transformation T of the records e . At the same time, M, being a confidence module, cannot affect its environment except by writing into b_2 -branded variables, (by r_3 , any real number can be converted to a b_2 -branded object). A b_2 -variable, in turn, can only be written on E' , according to r_4 . No other operation can be applied to it.

This solves our problem; the programmer can carry out an arbitrary transformation on E, storing the results of his transformation on E' , but he has absolutely no way to see the file E' itself or any information about it, or to leak such information to anybody else. At the same time, the program Π as a whole is not confined or limited in any sense. Of course, this solution depends on the availability of confidence-modules.

(1) For simplicity we are assuming that the records $e \in E$ are simply real numbers.

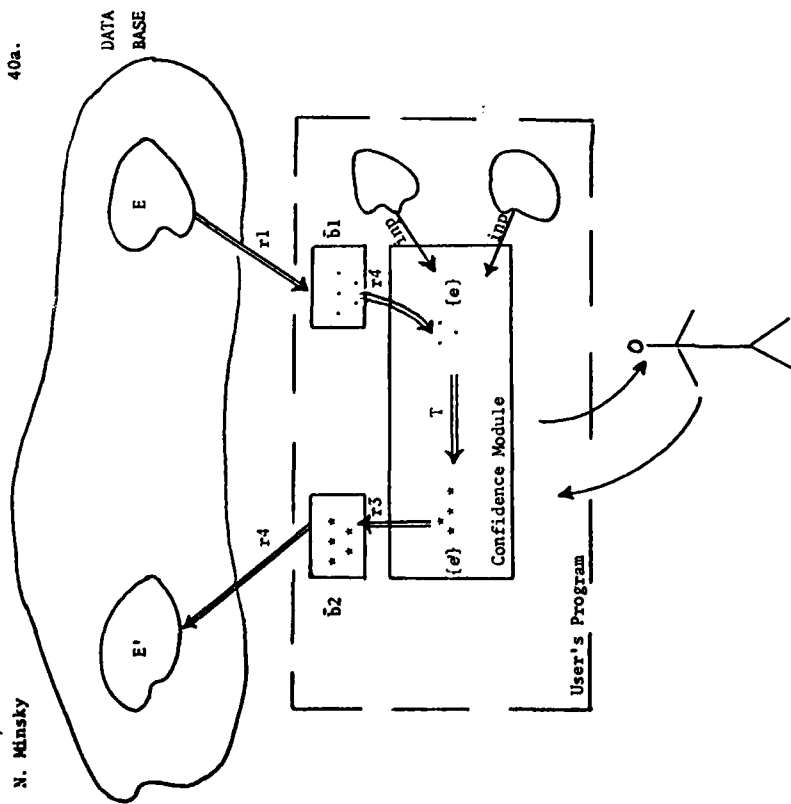


FIGURE 4.1

This figure illustrates the situation created by the subschema I of example 4. The records ecE which are retrieved in an opaque box, $b1$, which shields them from Π . (The box represents here the brand $b1$). The only way out of this box leads into a confidence-module, which in turn has only one output channel which leads into another confinement, the box $b2$. Within the confidence module M , the records ecE can be manipulated freely so that an arbitrary transformation T can be performed on them. The single arrows labeled $r4$ illustrate the fact there is no restriction on input channels to a confidence module. The labels $r1 \dots r4$ on the double arrows represent the D-rule which legalizes the transmission of information represented by the arrow.

5: Comments on the concept of "type"

The reader may have the feeling that our concepts of *brand* and *application rules* are nothing but a crude attempt to reinvent the good old "type" concept. It is, therefore, worthwhile to put things into proper perspective.

First, the main purpose of our "capabilities" model for programming languages, introduced in section 3.2, was not to provide a better "typing system" for the language itself. Rather, our objective was to facilitate the amplification of the language by the capabilities of Σ . For that purpose it was convenient to isolate the "types" and the "application rules", which are normally implicit in the syntax and semantics of the language. In addition, however, we believe that the model of section 3.2, has some merits even for a language which is not used for interaction with data bases, as we will try to show next.

In spite of the significant variations among researchers as to their approach to "types", they all seem to agree with the following quote from Liskov and Zilles [4]: "An *abstract data type* defines a class of abstract objects which is completely characterized by the operations available to these objects". Thus, "typing" is a classification of objects, and at the same time, a specification of how each class can be manipulated.

In programming languages, these two aspects of "type" are usually inseparable; in our model, on the other hand, they are deliberately separated: the classification of the data objects is done by means of the *brands*, while the specification of how they can be manipulated, is contained in the *application rules*. This separation provides us with flexibility which might be useful in a number of circumstances. Some of them will be mentioned below.

As in the case of operating systems, it is sometimes desirable to give to different modules of a program different "access-rights" to certain objects. For example, one might want a certain block to have a "read-only" access to a given global variable. This cannot be done in most languages since the access rights to an object are rigidly associated with the object itself. If an object q is at all accessible to a given module M , this module can usually apply to q any operator which is defined for it. (This rigidity is not removed even by some of the newest approaches to the concept of type, such as the work of Morris [10], for example). However, no such difficulty exists in our case, since every module may have its own application rules. (1) This, of course, is analogous to the "capability model" of operating systems (see Lampson [11]), except that our "application rules" are more general, and somewhat different from the "capabilities" of operating systems.

Another interesting possibility, facilitated by the separation between brands and application rules, is the ability to manipulate the brands themselves. The following is an example of one case in which this might be desirable.

Quantities which appear in physics have a *dimension* associated with them. This dimension can be represented by the three real numbers l, m , and t , which are the powers of the basic units of *length*, *mass* and *time*, in the given quantity. Now, it is well-known that physical formulas must be "well formed" with respect to its dimensions (see below) in order to be correct. This requirement serves physicists as a powerful tool for checking the validity of their formulas. It should be a very useful restriction to impose on their programs as well. In other words, we want the programming language to guarantee the well formedness of all formulas in a program.

-
- (1) Of course, this would require a generalization of our "simple application rules".

This can be done essentially⁽¹⁾ as follows:

Let us associate a brand $\tilde{d} = (l, m, t)$ with every "real" number which appears in a program, which is simply the *dimension* of this number. Let us impose the following "application rules" on the program:

$$r1: \tilde{d1} + \tilde{d2} = (+, \tilde{d1}, \tilde{d2})$$

$$r2: \tilde{d1} - \tilde{d2} = (/ , \tilde{d1}, \tilde{d2})$$

$$r3: \tilde{d1} = (+, \tilde{d1}, \tilde{d2}) / (\text{if } \tilde{d1} = \tilde{d2})$$

$$r4: \tilde{d1} = (-, \tilde{d1}, \tilde{d2}) / (\text{if } \tilde{d1} = \tilde{d2})$$

Rule $r1$ states that any two quantities can be multiplied by each other, and that the brand (dimension) of the outcome of such multiplication would be the sum of the brands of the two operands. Rule $r2$ has a similar nature.

Rule $r3$ (and similarly $r4$) states that two quantities can be added to each other, only if their dimensions (brands) are equal, the outcome would have the brand of the operands.

The enforcement of these rules guarantees the "well formedness" of all⁽²⁾ formulas in the program. Moreover, there is a potential possibility to do some of this enforcement at compile time. The user may point out that similar results can be achieved if every physical quantity is represented by a pair $(value, \tilde{d})$, and if the basic arithmetic operators are appropriately defined for such pairs. Such solution is less convenient, however. In particular it does not allow any compile time checking.

It is hoped that this very incomplete discussion is sufficient to convince the reader that our particular way of dealing with concept of type has some interesting possibilities.

-
- (1) Many details are left out of the following discussion, for brevity.

- (2) This is not strictly correct, but due to space limitations we cannot go here into the fine details of the problem.

6: Conclusion

In conclusion we would like to examine the main implications of this paper to the design and use of data base systems. First, most of the discussion above is directly relevant to what is commonly called "data manipulation language", (or DML). The concept of DML was introduced, by the DBTG report, as an extension to a "host language", which enables its use for interaction with a data base. This extension, as it is viewed by the DBTG report, is essentially an addition of new linguistic constructs to the host language, and it is usually achieved by means of a preprocessor which translates the new DML-constructs into procedure calls in the host language.

This paper suggests, however, that the data-manipulation-language is not an "add on feature". First the extension of the host language, which we called the amplification of L by I, is a more subtle process than what the DBTG had in mind. Furthermore, this paper presents some strong requirements to the host language itself. In particular, the host language (or what we called the "base language") must have sufficiently powerful application-rules, which must be strictly enforced. This requirement immediately rules out the use of a machine language for interaction with data bases. Moreover, even higher level languages, like COBOL or FORTRAN, cannot be used as base languages, primarily because they usually do not strictly enforce their own rules, but also because the rules they have are not powerful enough. An example of a language which is better qualified for interaction with data bases is SIMULA, particularly with the newly proposed protection features for this language [8]. Another work which should be mentioned at this point is a proposal to extend PL/I into a DML [9].

One serious objection that the reader might have to our approach to protection, is that we are spreading the protection mechanism to thin: A protection mechanism is of little value if one cannot be reasonably sure that this mechanism itself works correctly. For that it must be small because nobody knows how to prove, or completely debug, a large system. This observation led several researchers [2,13] to suggest that the protection mechanism of a system must be its kernel. This paper, on the other hand, pushes parts of the protection mechanism to the periphery, essentially to the compilers of the languages used for interaction with the data base. Now, can we rely on such a big system as a compiler, and in fact on several of them?

One answer, to this very serious objection, is that we do not seem to have much choice. It is, in fact, the main result of this paper that the user's program must be monitored, if the data base is to be protected adequately. In particular, intentional resolution of privacy protection, which has a great practical importance, cannot be achieved otherwise. But there is also another answer, which is related to the second major implication of this paper.

Up to this point, the "user" was assumed to be an external user; actually, however, the term "user", in this paper, admits a broader interpretation. The data base, as any other large system, is likely to be modularized; and about the same protection problems which exist between a DB and its external user, exist also between the DB and its own modules. Such a module, which may be one of the D-operators for example, should have only a partial, and possibly abstract, view of the data base in which it is embedded. Thus, all that has been said in this paper about the user's program, is also applicable to such a module. This observation suggests that the data base itself should be built, or programmed, by means of a language which has the necessary protection

features, it may be called the DB-language (or DBL). This DBL should play the role of the base-language in our model, in the following sense: To design a module of the DB, say a D-operator, one has to work under a certain Σ , just like any external user. The module would therefore be written in the Σ amplified DBL, namely DBL/ Σ .

Coming back to an external user, which has to operate under a given Σ . We might be able to allow him to write in an arbitrary programming language, provided that his program is translated into DBL/ Σ . Thus the data-base language, in which the data base is designed and under which the external processes which interact with the DB operate, is in a sense the protection kernel of data-base system. (This way of looking on the data base and its protection, needs further study).

Finally, it should be pointed out, that this paper should not be considered as a specific proposal for a DML, or for a protection mechanism for data bases. Our purpose was only to indicate some of the problems posed by the process of user-database interaction, and to suggest an approach for their solution.

References

- 1) CODASYL Data Base Task Group (DBTG) April 71 report (Available from ACM).
- 2) Minsky, N., "On interaction with data bases" Proc. of the ACM SIGFIDET workshop "on data description access and control", 1974.
- 3) Codd, E.F., "Relational Completeness of data base sublanguage". Proc. of Courant Institute Symposium on data base systems, 1971.
- 4) Liskov, B. and Zilles, S. "Programming with abstract data types", Symposium on very high level languages, March 1974.
- 5) Conway, R., W. Maxwell, and H. Morgan, "On the implementation of security measures in information systems" Communication of the ACM, April 72.
- 6) Minsky, N. "On the 'resolution power' of privacy protection of data base systems" Dep. of Comp. Sci., Rutgers Univ. Tech. Rep. SOSAP-TR-9 June 1974.
- 7) Lampson, B. W., "A note on the confinement problem" Communication of the ACM, October 1973.
- 8) Palme, J. "Protected program modules in SIMULA 67". FOAP report C 8372-M3 (E5) July 1973.
- 9) Summers R.C., Coleman and Fernandez, "A programming language approach to secure data base access", IBM Tech. Rep. G320-2662, May 1974.
- 10) Morris, J. H., Jr. "Protection in programming languages", Communication of the CAM, Jan. 1973.
- 11) Lampson, B. W. "Protection", Proc. Fifth Annual Princeton Conf. on Inf. Sci. and Systems, March 1971.
- 12) Popek, G. "A note on kernel design", AFIPS Conf. Proc., 1974 NCC AFIPS press.
- 13) Rulf, W. A., et al. "HYDRA: the kernel of a multiprocessor operating system". Communication of the ACM, June 1974.

ANOTHER LOOK AT DATA-BASES

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

ANOTHER LOOK AT DATA-BASES*

Naftaly Minsky
Dept. Of Computer Science
Hill Center, Busch Campus
Rutgers University
New Brunswick, N.J. 08903

1. What is a Data Base?

A data base is often viewed as a kind of glorified file which allows for the storage of complex data structures, and which can be shared by many users who have different needs. But there is a more fundamental difference between data-bases and files than just the complexity of data or the generality of its retrieval. The traditional file is essentially a passive collection of coded data which is devoid of any intrinsic meaning. Indeed the interpretation of the data stored in a file is entirely up to its users. The data-base, on the other hand, is usually used as a model, or representation, of some system in the real world. For example, a corporate data base is, in some sense, a model of a corporation; and it must reflect some of its structural and behavioral properties. This is not just a matter of recording correctly everything which happens within the corporation. For example, when we "tell" the DB that an employee is fired, we expect all necessary adjustments to be made automatically, so that only current employees will be on the payroll of the corporation at every moment in time. In short, we expect a DB to "take care of itself" to a certain extent; it therefore cannot be a completely passive collection of data. But if a data-base may contain procedural components and not just data, what kind of thing is it? In what way is it different from what we usually call programs? (It might have never occurred to the reader to compare data-bases with programs, but as we will show later the two concepts have more in common than meets the eye.) In an attempt to clarify the concept of data-base I will now offer a definition for it. The definition is an oversimplification, as any definition of such a fuzzy concept must be; it will emphasize certain characteristics of data-bases, at the expense of other, equally important, properties.

Definition: A data-base is a model (or representation) of some system in the real world, which satisfies all of the following properties:

- (a) The model contains a large amount of coded information.
- (b) It has a long life time (say, from several days to several years).
- (c) The model can be examined and interrogated at any moment of its life time.
- (d) The model changes primarily in response to operations applied to it from the outside.

*This work was supported by Grant # DAHCIS-73-G6 of the Advanced Research Projects Agency.

**Best
Available
Copy**

The first sentence of this definition states the general objective of a data base. This objective is not specific to data bases, however. For example, a corporation can be modeled by a simulation program, not just a DB. The specificity of this definition to data-bases is provided by properties (a) to (d), the first of which we assume to be tacitly understood.

To see how the other properties relate specifically to data-bases consider again a corporate DB. Such a DB functions as a model, or a representation, of a corporation for long periods of time. At every moment during this time the model can be interrogated, which corresponds to retrievals of information from the DB. Note that this is quite different from the behavior and usage of traditional programs.

A program is used primarily as a kind of input-output engine; it is executed for its output, and it is not usually examined while in the process of computation. Indeed, the active period of a program, what may be called its life time, is relatively short and insignificant.

Another difference between data-bases and programs is provided by property (d) which states that a DB changes primarily in response to operations from outside, update operations if you will. This is in sharp contrast with, say, a simulation model of a system whose dynamic behavior is determined primarily by the model itself. This property also introduces a new dimension into the sufficiently difficult problem of correctness in programming systems. When we write a program we have to make sure that the program will do exactly what we want it to do, which is difficult enough. But the designer of a DB has an additional problem at hand. He does not know in advance how the DB will change in time; this is up to the users who will interact with it. At the same time, it is up to the DB designer to establish the invariance of certain structural and behavioral characteristics of the DB whatever the users of the DB might do. In other words, the integrity of the DB must be protected against the essentially unpredictable interaction with the outside world. (Of course, this problem is not peculiar to data-bases, it exists to a certain degree in many programming systems, particularly in operating systems; but the need for protection is particularly acute in the case of data-bases.)

Most of the current research on data-bases is directed towards various implications of properties (a) to (d). People are most concerned with subjects like the physical and logical representation of complex data, techniques and languages which allow flexible retrieval from the data-base, etc. But it seems that the objective of data bases, to be a model of a system in the real world, is somewhat discarded, or at least, is not sufficiently emphasized. True, the protection of the integrity of data bases is often stated as one of the main objectives of data base systems. But in most cases this is hardly more than lip service, supported, at best, by simple consistency checking. But if there is a lesson to be

learned from the long experience with programming in general, and with operating systems in particular, it is that protection is not an add-on feature but must be designed deeply into the fundamental structure of a system. This, to the best of my knowledge, is usually not done in the case of data-bases.

In this paper we will examine some aspects of the architecture of data-bases having in mind primarily the need to protect their integrity with respect to the real-life system which they are supposed to represent. The conclusions of this paper are not new, most of them are borrowed from ideas and principles developed in the context of operating systems and programming languages. If there is any originality in this paper it is the application of these well known principles to data-bases.

To remove any possible misunderstanding it should be pointed out that we are dealing here with the concept of data-base not data-base management system (DBMS). The relationship between these two concepts is somewhat analogous to the relationship between a program and its compiler.

2. A Data-Base-Language

It is well known that the structure of programs is strongly affected by the properties of the programming language in which they are written. A similar statement is probably true for data bases as well. Most of this paper will, accordingly, be concerned with the language used for the construction of data-bases. We will begin by reviewing the language proposed by one of the most influential works on data bases, the Data Base Task Group (DBTG) report [1].

Two languages are introduced by the DBTG report: The Data Definition Language (DDL), which is a declarative, non-procedural language, to be used for the definition of a DB; and the Data Manipulation Language (DML) to be used for interaction with an existing DB. This dichotomy of languages, which is somewhat analogous to the sharp distinction made in COBOL between the "data division" and "procedure division", seems to imply that data-bases are essentially passive structures, since they are to be defined by a language which does not feature any data-manipulation capabilities. This approach certainly does not agree with our view of a data-base as a model of a possibly dynamic system. Moreover, it is not even compatible with the modern approach to data structures in general, as we will see later. Indeed, the DBTG report did not really view data-bases as completely passive structures. To compensate for the lack of procedural capabilities in its DDL, it allows for the incorporation of the so called DB-procedures within the data base. But the report did not specify how such procedures should be incorporated into a DB, or in which language they should be written (should it be a DML)? These questions are generously left for the implementor, in spite of the fact that the declarative features of the DDL were described in great detail. It appears that the authors of the DBTG report considered the DB-procedure just as an additive feature of marginal importance.

It is my contention, however, that the language to be used for the design of data-bases must be an integration of declarative and procedural capabilities, as most programming languages are; one

may call such a language a Data-Base-Language (DBL). An obvious reason for having such a language is that if procedures are at all necessary for the design of data-bases, there seems to be no good reason to introduce them via the back door, as is done by the DBTG report. But there is a more subtle reason for that. The DB-procedures are the dynamic components of a DB. If we want to have any control over the dynamic behavior of the data base we must be able to control the DB-procedures themselves. In particular, one should be able to restrict a given procedure as to what it can do to the rest of the DB. It appears that the clearest way to achieve that, is to have all DB-procedures be written in a given, well-defined, DBL. The manner in which such a language can be used in order to impose certain disciplines on the behavior of a DB, will be discussed in the rest of this paper.

3. "Information Objects" in Data Bases.

What was said about the data-base being a model of some system in the real world should be true also for the individual components, or records, stored in a DB. At least some of them must represent specific objects in the real world. For example, we might have a data structure which represents an employee in a corporation, and another one which represents a job. The behavior of these data structures within the DB must reflect, in some way, the behavior of the objects they represent within the corporation itself. For instance, one should be able to hire or fire an employee, or to assign him to a certain job; but one should not be able to manipulate arbitrarily a data-structure which represents an employee, or a job. Thus, it seems that we need the ability to form data structures which can be manipulated only in a certain predefined way. Data structures which have this property will be called here *information objects*.

Note, however, that being an information object is not really a property of the data structure itself, but of the "computing environment" in which it exists, which should prevent any illegal manipulation of the object. Such a computing environment can be created by a programming language, as was demonstrated by a number of researchers.

In a recent paper by Liskov and Zilles [3], a language is described which features a linguistic construct called *cluster* which serves as a template for a class of information objects. The cluster contains a description of a conventional data-structure, together with a set of procedures which are defined on that structure. For a given cluster *c*, there is a way to generate objects using *c* as a template. All such objects are said to belong to class *C*. Now, the crux of the matter is that an object of class *C* can be manipulated only by means of operators defined within the cluster, which turns these objects into information objects, according to our definition.

Note that this is not an easy proposition; it is not just a matter of adding yet another feature to an existing language. The entire language must be very carefully designed in order to guarantee that

an object of class C is never manipulated by an operator which does not belong to it. Note also that the combination of both declarative and procedural statements in the same cluster is necessary for the formation of these information objects.

We will now try to demonstrate, by an example, the potential usefulness of information objects for data-bases, assuming that this facility is supported by the DBL at hand.

Example: Conservation of Money

One type of object which has to be represented in any financially oriented DB, is money. One of the most important properties of real money is that it cannot be created out of nowhere, at least not in a legally operated establishment, nor should it be allowed to disappear into thin air. This property may be called *conservation of money* and it should be reflected in the behavior of the data-structure which represents money in a DB.

To impose conservation of money on a DB, we will represent money by means of information objects to be called *safes*. A safe is a kind of money-variable which can be accessed and manipulated only* by means of the following operators:

Content(s) returns the number of dollars currently stored in safe *s*.

move(s₁, s₂, k) if *s₁, s₂* are safes and *k* is a positive number such that $\text{content}(s_1) \geq k$, then this operator "transfers" units of money (dollars) from *s₁* to *s₂*. (Namely, it leaves the content of *s₁* smaller by *k*, and that of *s₂* larger by *k*).

The operator *move* represents the act of money changing hands in the real world. In order to represent the flow of money into the system we use another class of information objects, to be called *source*. A source is similar to a safe except that any amount can be moved from a source into a sink, but no money can be moved into a source. The content of a source would be negative, its absolute value being equal to the total amount of money moved out of it.

Under these conditions, and assuming that the content of all sources and safes is initially zero, it is easy to see that the sum of all safes and sources in the system is always zero.

In spite of the obvious usefulness of being able to impose such a law of conservation on a DB, it is not quite sufficient. For example, one would like to impose certain disciplines on how the money flows between the various safes, and how it is used within the DB. We will return to these problems in the next section.

*Except for operators which generate and destroy safes, which are not discussed here.

4. Environments of Execution

As was just demonstrated, the ability to create information objects which can be manipulated only by means of given operators can help in enforcing useful disciplines on the behavior of a DB, but this ability alone is certainly not sufficient. One should also be able to specify "who" can apply which operator to a given object. Such a capability is fundamental to the various protection schemes recently developed for operating systems (2), it also forms the basis for many of the techniques used for the protection of data-bases against users who interact with it. For example, according to the DBTG proposal, a DB-user operates within an "environment" formed by a sub-schema which specifies, in effect, which parts of the DB the user is allowed to get, and how he is allowed to access them. But although this does provide a DB with certain protection against its users, it does not protect it against itself.* A DB is manipulated not only by its users, but by its own procedures as well; and it is a well known principle in the design of large systems that no module should be allowed to access more of the system than it needs, in order to perform its function. Thus, the DBTG proposal should have assigned a sub-schema to every DB-procedure, in order to limit the procedure as to what it can do to the rest of the DB. This kind of protection scheme was proposed by Wulf, Jones and others (2), for the design of operating systems. Although their techniques must be modified before they can be applied effectively to data bases, the basic philosophy has universal validity, and is briefly described below with some changes, and slightly different terminology.

According to Jones' and Wulf's proposal, every activity (or what is usually called "process of computation") in a system is carried out from a certain environment E , which can be formally described as a set of pairs $E = (q, a)$, where q is the identifier of an information object in the system, and a is one of the access-modes defined for it. The pair (q, a) is called *capability* (or *right*). A capability $(q, a) \in E$ serves as a right for a process which operates under E to exercise access a to object q , in the following sense. Every operator defined as an information object in the system, expects the environment from which it was invoked to have certain capabilities. Namely, the operation is carried out only if the environment in which the requesting activity resides contains the required capabilities. An example may clarify this point.

Consider again our money example. Let *get* and *put* be two access modes defined over a safe such that the operator *move* (s_1, s_2, k) can only be carried out if the calling environment has the capabilities (s_1, get) and (s_2, put) . This allows us to specify not only which safes a given user can access, but the direction in which he can move money between them.

One of the most interesting aspects of Jones' model is its treatment

*It should be pointed out that even in this respect the sub-schema of the DBTG report has serious deficiencies, as is shown in (4).

of procedures: Every procedure is defined within its own environment, which obviously does not depend on the environment of the potential caller of the procedure. The capabilities in this environment are called the *static* capabilities of the procedure. In addition, when the procedure is called, it may receive some additional, *dynamic* capabilities, from the caller's environment, by means of its parameters.

It is quite clear that such protection schemes can greatly enhance the reliability of a large programming system. But I would like to demonstrate specifically the usefulness of such a scheme in the context of data bases, using for that, our money example.

In the previous section we saw how to establish a global "conservation of money" in a corporate DB. But more than that is required in order to model the behavior of money in the real world. For example, assuming that paychecks are issued under the direct supervision of the corporate DB, one may require that whenever a check is issued on the amount of D dollars, the total amount of money represented in the DB is reduced by D . The implementation of this and other characteristic properties of money are discussed below.

Consider a DB-procedure $PAY(e, s)$ which is designed to issue a check payable to employee e for the amount contained in safe s . The following assumptions are made.

- a) PAY has a static and exclusive put right to a safe called *cash*, (namely, it has the capability (*cash*, *put*) in its environment).
- b) No environment in the DB has a *get* right to *cash*. (The safe *cash* is, therefore, a *sink* of money).
- c) PAY has a dynamic *get* right to the safe s given to it as a parameter, and it has no other right to any other safe.
- d) PAY is the only procedure which is able to actually print a check.

Now, if the procedure PAY is known to perform the operation *move* ($s, cash, content(s)$) after printing a check for the amount of *content(s)*, then we are assured that every check printed by the DB is balanced by a suitable amount of dollars being transferred to the safe *cash*. Moreover, *content(cash)* is equal to the total amount of dollars which was paid by checks.

This example can be further refined to establish *local* conservation of money, as follows: consider an environment E , which may be the environment of some user, or of a DB-procedure. Suppose that E contains the *get* right to a single safe s_0 , and both the *get* and *put* rights to the set of safes s_1, \dots, s_n . Suppose also that no other environment has the *put* right to s_1, \dots, s_n , and that E has the right to invoke the PAY procedure. E may, for example, be the environment used by the clerks in the accounting office of a certain

department in the corporation. It is clear that whoever operates under E, has only as much money available to him as he can get from s_0 , which is his only external source of money. For example he can never pay more in checks than what is given to him via s_0 . Thus, s_0 can be used by an environment which happens to have the put right to it, as a way to allocate budget for E. Similarly, E may grant some of his own money to some other environment via one of the safes s_i .

It is clear that by these kind of restrictions on the manipulation of safes one can model many of the properties associated with money in the real world. A monetary information system built in such a way also lends itself very nicely to auditing. The auditor can know much about the flow of money inside the corporation just by examining the various environments and some DB-procedures such as our PAY procedure.

The implementation of the protection scheme described above is beyond the scope of this paper. I wish only to suggest that it should be entrusted to the Data-Base-Language. This language has a unique position from which it can control what happens within the DB, because the DB itself is presumably defined in it. In this sense, the DBL will play the role of the protection kernel of operating systems proposed by Wulf and Jones. (For the reader who is puzzled by the idea that a language can be used for protection, I would like to point out that in a certain sense a programming language can be viewed as a set of capabilities which are provided to a program written in this language. This view about languages, and its implications, to the ability of a language to play a major role in protection is discussed in (4).

Conclusion

A DB has been viewed as a dynamic model of some given system, rather than as a passive collection of data. This paper was concerned primarily with some implications of this point of view to the language used for the construction of data-bases. One obvious implication is that a Data-Base-Language must have procedural capabilities. A somewhat more subtle observation made by this paper, is that a DBL should have certain features which would enable a DB designer to impose some discipline on the behavior of his DB, under its interaction with users. There are, however, other important issues concerning the Data-Base-Language, which were not discussed here. In particular, there is the problem of the control-structure of such a language. There are indications that the conventional control-structure of, say, ALGOL-like languages, would not be sufficient. For example, one needs tools to cope with the parallelism in the activity of data-bases. In addition non-conventional control structures such as "event-sequencing" (or "demons"), which were proposed for data-bases by Morgan (5), and which are being used in several AI-languages, appear to be very promising. These and other issues must be investigated in the context of the already existing knowledge about the structure and manipulation of complex data (1,5).

ON INTERACTION WITH DATA BASES

N. Minsky

Department of Computer Science
Rutgers University
New Brunswick, New Jersey

ON INTERACTION WITH DATA BASES

By Naftali Minsky

"When I use a word," Humpty Dumpty said
...it means just what I choose it to mean
- neither more nor less." "The question is,"
said Alice, "whether you can make words
mean so many different things."

"The question is," said Humpty Dumpty,
"which is to be master?"

Through the Looking Glass
by Lewis Carroll

Introduction

There is a fundamental difference between *data base* and *files* which is not just a matter of size or complexity. A file is essentially a meaningless collection of bits whose interpretation is entirely up to its user; the data base, on the other hand, has an intrinsic meaning and structure which is independent of its users. This characteristic of the data base creates quite an unusual situation for its users. Programmers usually operate within a fairly isolated system which they control. The programmer defines his own information space, and as long as he conforms to the general framework set up by the programming language at hand, he can manipulate his information as he pleases. On the other hand, a data base user operates within an environment which is only partially known to him. Moreover, any interaction of a user with a data base must be performed on terms set up by the data base, otherwise its intrinsic structure might be endangered.

In this paper we will study the implication of this, somewhat unusual, state of affairs to the process of user-data base interaction (to be denoted by *U-D interaction*).

1. The Data Base Rules

It was stated above that the fundamental property of a data base is that it has an intrinsic meaning which is invariant of its interaction with users. The significance of this statement may be clarified by an example.

Let D be a school data base which contains, among other things, the class schedule of the school. This can be represented by means of a relation (class, room, time) which gives the time and location of every class. The users of the data base would probably expect the class-schedule to satisfy various rules. For example, there should be no conflicts in the

ON INTERACTION WITH DATA BASES

schedule, no student should be forced by the schedule to participate in two classes which are given at the same time (assuming that there is also a class-student relation in D), etc. Now, in most conventional data base systems there is no formal way to state such rules; they are usually established, de-facto, by mutual agreement between the various users of the base. Obviously, however, the credibility of such rules is not higher than the credibility of the data base users as programmers.

Thus, it is clearly necessary for the data base to contain a formal specification of its various properties; such specification will be called the *data base rules*, or *D-rules* for brevity.

One may distinguish between two types of D-rules: *integrity rules*, and *privacy rules*. The *integrity rules* are concerned with the well-formedness of the data base. These rules may have various forms and types. Here are some examples:

a) *Domain rules*, which specify the domain of various data objects.

b) *Consistency rules*, which specify relations that should be satisfied between objects. (For example, the rule which states that the class schedule should not have conflicts.)

c) *Activation rules*, which state that a certain procedure should be activated when a given situation occurs.

The *privacy rules* are concerned with what various users can do to the data base, and what information can they get from it. (The structure of such rules is discussed in [3].)

It is obviously not enough to have a formal specification of D-rules; one must be able to enforce them. In this paper we will look into some implications of this basic requirement to the way in which users should interact with the data base. Our discussion will be fairly general in the sense that it will not depend much on any particular type or representation of the D-rules. This, however, does not mean that we will come up with a general method for the enforcement of D-rules; we will only be able to formulate some necessary conditions for the D-rules to be enforced.

2. User's view of the data base

One of the most important characteristics of the user data base interaction is that the user usually has an incomplete knowledge about the data base on which he operates; moreover, different users may have different views of the same data base. There are several reasons for this. First, due to the typically large size of D, and because its content is usually generated by many independent users, no single user can be expected to be familiar with the entire data base, or to be interested in it. Secondly,

¹ We will use the letter D to denote a data base.

certain parts of the data base may have to be deliberately hidden from a particular user because of security reasons.

This is not just a matter of providing the user with a partial view of the data base in which some of its parts do not appear; it should be possible to present the user with various levels of abstraction of the various parts of the data base. For example, in an inventory data base, a "material-item" which is characterized by several attributes may appear to some users as an object with, say, the attributes name and price only, and which can be ordered and received by such a user. It is important to realize that an abstraction of an information object is not just a less-detailed image of it. An abstraction may contain "things" which do not exist in the original object, such as operators which are applicable to the abstraction only. Examples of such operators will be given in the next section.

Thus, the user U does not interact with the data base D directly but with an image of it, to be denoted by \bar{D} . Following the DBCG report [1] we will use the name *sub-schema* for the definition of an image \bar{D} of D . The sub-schema is itself part of the data base, moreover there would usually be many different sub-schemas in D , built for various classes of users and various types of use. In order to interact with the data base, the user U must attach himself to one such sub-schema. The image \bar{D} defined by this sub-schema becomes in effect the "virtual data base" from the point of view of the user, it will sometimes be denoted by $\bar{D}(U)$. In this paper we will not discuss the structure of the sub-schema nor how it is attached to a user, (some of this is discussed in [3]); we will only be interested in the implication of such an indirect interaction between users and the data base.

It should be pointed out that the notion of sub-schema was first introduced by the DBCG report [1], primarily in order to facilitate "data independence" of user's programs. But the image \bar{D} generated by their sub-schema is almost literally partial to D ; namely, the sub-schema of the DBCG report can hide parts of D from the user, by not including them in \bar{D} ; but it cannot form abstractions. This is a very serious limitation. The ability to present a user with an abstract image of the data base which is suitable for his needs is not only desirable from the point of view of "software engineering"; as we will show next, this ability is necessary for the security of the data base.

3. The Primitives of User Data Base Interaction

In this section we will discuss the operations which should be available to users for their interaction with the data base. Conventionally, these are essentially *update* and *retrieval* operations. Although some data base systems feature very sophisticated techniques by which one can select and identify parts of the data base, the action performed on those parts is practically always either update or retrieval. While this update

and *retrieval* oriented approach is natural for files, it is not suitable for data bases, as the following analogy might illustrate.

The driver of a car is usually unfamiliar with its internal structure, what he has in his view is an "abstraction" of this structure: the gear box is abstracted by the gear shift lever, and the two front wheels by the steering wheel. The driver cannot directly change (update) the state of the various parts of the engine, he must use for that the special "operators" which exist in his view. For example, he does not have a direct access to the front wheels; to turn them he must use the steering wheel. This is not only more convenient, it is also safer. By forcing the driver to use the steering wheel one guarantees that the two front wheels are always turned in the same direction.

The analogy to data bases is obvious: Like the car driver, the data base user sees only an abstract image of the data base. It is clearly impossible for him to directly update an object whose detailed structure is not known to him. For the manipulation of such an object one needs special abstract operators which are suitable for the specific image of the object. Moreover, like the car, the data base is a structured mechanism which must behave in a certain way. Our analogy clearly suggests that the use of predefined, high level operators may help to enforce the proper behavior of the data base. This conclusion will be now substantiated by showing that certain D-rules cannot be guaranteed without the use of such operators as primitives of U-D interaction.

Consider the set of objects $e_1 \dots e_k$ stored in D , and let us formally define e to be the k -tuple $\langle e_1 \dots e_k \rangle$. (e is defined for the sake of discussion only; there is no implication that such a structure exists explicitly in D .) Let r be one of the D-rules which is defined over $\{e_i\}$, and which, in effect makes some of the k -tuples $\langle e_1 \dots e_k \rangle$ to be illegal (namely, some of them do not satisfy r). The question to be discussed now is how does one guarantee that only legal states of e will be stored in D . This obviously depends on the form of the rule r , or more specifically, on the way in which the legal states of e are specified. We will consider two such specification methods, to be called *functional* and *constructive*.

The functional specification of the legal states of e is a predicate $r(e_1 \dots e_k)$ which has to be satisfied for every e stored in D . (For example, such a predicate may state that the sum of all the expenditures of a department in a corporate data base cannot exceed its budget.) Apparently, such a rule can be very easily enforced: whenever an e_i is updated, the predicate can be re-evaluated, rejecting the update operation if it falsifies r . However, there are several problems with this method.

First, if the update of every e_i is considered to be a primitive operation, and should therefore be checked for legality, the user may not be able to update e at all. To see that, consider the case $k = 2$ and suppose that the

only legal state of e are $\langle e_1, e_2 \rangle$ and $\langle e_1', e_2' \rangle$, when $e_1 \neq e_1'$. Beginning with $\langle e_1, e_2 \rangle$, the user would not be able to change it to $\langle e_1', e_2' \rangle$; since every primitive operation, namely the update of either e_1 or e_2 , results in an illegal state. Thus the update of individual objects cannot be the only type of primitive available to the user.

Another difficulty with the functional specification is that its enforcement may be extremely inefficient, mainly because $r(e_1, \dots, e_n)$ has to be reevaluated every time any e_i is changed. Such enforcement technique can easily have a catastrophic effect on the performance of the data base system, and is, therefore, unrealistic in many cases.

Let us next describe the constructive specification of the legal states e . Let P_1, \dots, P_n be operators, each of which transforms a k -tuple $\langle e_1, \dots, e_k \rangle$ to another such k -tuple by changing one or more of its components. The legal states of e can be defined by specifying the "initial legal state" $\rightarrow e_0$, and by the following rules: If e is a legal state, so are $F_i(e)$ for $i = 1, \dots, n$. (For obvious reasons, $\{P_i\}$ will be called the legal operators.)

The usefulness of such constructive specification, when it is applicable, is obvious: if the user is forced to use only legal operators on e , he cannot, by definition, create any illegal state. Moreover, if the constructive specification is the only one available, then one may not be able to guarantee the legal state of e if the user is not forced to use the legal operators. Because, if the user simply updates e to e' , it may not be possible to determine if e' can be produced from e by means of the legal operators. Thus, if a rule is given by means of the constructive specification, then the use of operators as primitives for interaction is not only more convenient and efficient than direct update and retrieval, in some cases it is also vital for the security of the data base. Namely, there are rules which cannot be guaranteed unless the user is forced to use only legal operators.

It should be pointed out that the constructive specification of D-rules is not always applicable, but in many cases it turns out to be the most natural, or even the only available representation. Frequently, the simplest way to maintain a coherent behavior of a complex object within a data base is to design a set of operators, and to let others interact with the object only by means of these operators. By definition, these operators become the legal operators of the object.

As a final argument, let us point out that the difficulty is not only with "integrity rules," some "privacy rules" are also incompatible with the update and retrieval primitives: A user U may not be allowed to read or update any of the objects e_i . Yet, he may be allowed to perform a certain operation on them, such as a statistical analysis of $\langle e_i \rangle$. The point here is that if the only primitive operations are update and retrieval of individual objects e_i , then

since the user is not permitted to use any of them, there is no way for him to carry out the operation which he is allowed to perform. Thus, the rights of a user, in the privacy sense, cannot be even specified in terms of update and retrieval operations only.

To summarize this section: it has been shown that update and retrieval operations are not always suitable to be the primitives of U-D interaction; one needs, sometimes, higher level operators for that. Such operators, which will be called D-operators (or D-functions), would usually be specific to the image \bar{D} with which U interacts, they should therefore be defined by the sub-schema.

4. A model for U-D interaction

We are now in a position to discuss the process of user-data-base interaction as a whole: In order to interact with the data base D , the user U writes a program Π using a certain programming language L . L may be a special purpose, possibly interactive, language; or it may be a general purpose language such as COBOL, possibly with some extensions which enable it to communicate with the data base. (We will have more to say about the language L , later.)

The information space in which Π operates consists of two parts:

- The local storage space defined within Π , to be denoted by $S(\Pi)$ (or simply by S).
- The image of the data base defined by the sub-schema to which U is attached, it will be denoted by $\bar{D}(U)$ (or simply by \bar{D}). In particular, $\bar{D}(U)$ contains the D-operators which can be used by Π for his interaction with D . When such a D-operator is invoked it may have two effects: First, it may generate information into the storage space of Π . Such an information will be called the outcome of the D-operation, it would be stored in some variable $v \in S(\Pi)$. (This is a generalization of retrieval). Secondly, the D-operation may change the data base itself, which may be called the effect of the D-operation and is a generalization of update of the data base. To activate a D-operation one may need more information than just its name, this additional information may be called the parameters of the D-operation.

Now, how does one guarantee the privacy and integrity of the data base? According to the conventional approach (see, for example [2]), this is essentially a matter of access control. One simply has to monitor the D-operations invoked by Π , making sure that only legal operations are applied to D . This can be done partially at "compile time" (in a sense) by allowing Π to use only D-operators which belong to $\bar{D}(U)$. Sometimes the access-control requires "run-time" checking of the information which flows into the data base and from it, particularly when there are "value dependent" D-rules to be satisfied. In principle, all this can be done by some security enforcer mechanism which is built into the sub-schema and which acts as a filter between the data base and the user's program, (see illustration below.)

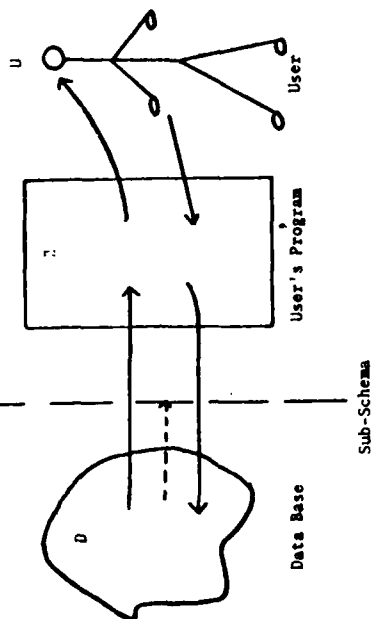


Fig. 1. The conventional approach to user data-base interaction. The sub-schema acts as a semi-opaque screen positioned between D and U, letting U see only selected parts of D.

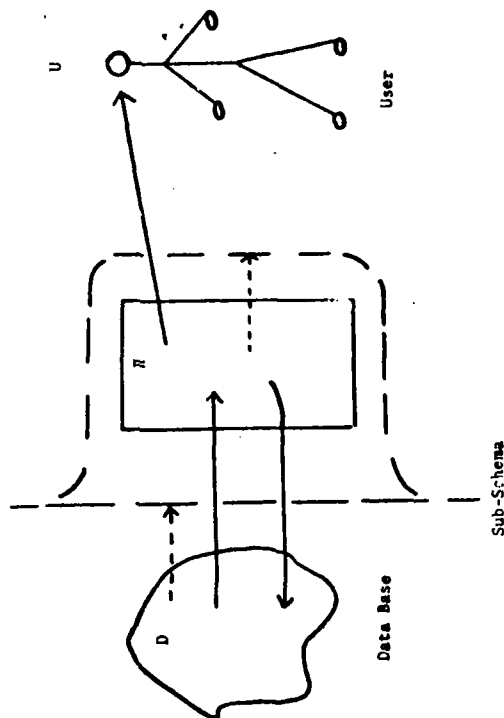


Fig. 2. An illustration of the proposed model for user data base interaction. This model distinguishes between the human user U and his program Π . Π is under a partial control of the sub-schema so that, for example, not every piece of information transmitted to Π is actually revealed to U.

In any case, the security of the data-base does not seem to impose any restrictions on the user program; accept that Π should be allowed to invoke only legal D-operations, which can be checked by the data base independently of what happens inside Π itself. This, in turn, means that Π can be written in any programming language, as long as it has the capability to invoke D-operations. In particular, it should be possible to write Π in assembly language.

These observations may seem to be so natural and even self evident that they are rarely spelled out, and yet they turn out to be incorrect. We will show in the next section that the security of the data base cannot be enforced by access-control alone, and that some restrictions must be imposed on the user's program itself.

5. Who controls the user's program

The issue to be discussed in this section is the affect of the data base rules on the user's program. It will be shown that in order to enforce certain types of D-rules it is necessary to impose some restrictions over what happens inside Π . This need will be demonstrated by means of several examples.

Example 1:

Let $F = \{f\}$ be a set of records containing, say, personal data about people; and let the attributes of f be $\{a_1, \dots, a_k\}$. Let U be a programmer who is commissioned to perform some statistical study on this data. For that he has to select a subset $F' \subset F$ which is relevant to his study, and then feed the selected records to a number of statistical procedures P_1, \dots, P_n which are provided to U by the data base itself. Now, for privacy reasons, one may want to prevent U from seeing the detailed structure of the records f . Assuming that the selection of $F' \subset F$ is based entirely on the attribute a_1 , U should be prevented from printing out the attributes a_i ($i \geq 2$) of the records $f \in F'$.

The difficulty here is that once a record $f \in F'$ is selected, and stored in $S(\Pi)$, it is automatically released to the user for any operation he might choose to apply to it. This suggests that there should be a way to restrict Π as to what it can do with an outcome of a given D-operation. A similar conclusion would be implied by the next example, for a fundamentally different reason.

Example 2:

Consider a medical data base D , and let P be a set of patient's names in it. Let a_1, \dots, a_k be D-functions defined over the members of P ; (for a given $q \in P$, $a_i(q)$ can be viewed as the i th attribute of q). Let U be a

doctor, and let $P'(U)$ be the set of all the patients-of-doctor U , ($P'(U)$ is a subset of P). Now, suppose that U is allowed to access the attributes a_1 of his own patients only. How can this privacy restriction be enforced?

First, one may try to secure the privacy of D against U simply by providing U with a partial image of the data base which contains only his own patients, as follows. Let $mypat$ be a D-function provided to U by the sub-schema to which he is attached. Suppose that $mypat$, when invoked by U , returns as its outcome the next name from $P'(U)$, (whatever "next" may mean). Assuming that this is the only way for U to retrieve elements from P' , our privacy rule may seem to be automatically satisfied. Unfortunately, this is not the case. Even if U can get only names of his own patients from the data base, he may still be able to get names of other patients from an outside source. Thus, U has the potential ability to invoke $a_1(q)$ when $q \in P'(U)$.

From the above it seems that the only way to guarantee that U does not get the attributes of a patient $q \in P'(U)$, is to check if q is actually a member of $P'(U)$ every time that $a_1(q)$ is invoked. This, however, is a very wasteful solution, it means for example that the membership of q in $P'(U)$ may have to be checked several times for the same q .

The nature of our difficulties would be better understood from a more general point of view, presented next.

Let f be a D-function, and v a variable in $S(\Pi)$. Suppose that f is invoked by the statement $v \leftarrow f$ which stores the outcome of f in v . In addition to the explicit information thus stored in v , it also carries an important implicit information which is the knowledge that the content of v is in fact an outcome of f . (For example, if a statement $q \leftarrow mypat$ is executed by Π of example 2 above, then q is known to be the name of a patient of U). Now suppose that v is used as a parameter of another D-function g . This communicates to g the explicit information in v , but g cannot rely on the implicit information in v since it has no way of knowing if the content of v was actually generated by the D-function f . Coming back to example 2, if $a_1(q)$ is invoked it is important for the data base to know whether q belongs to $P'(U)$, otherwise $a_1(q)$ cannot be carried out. This is indeed the case if q was generated by the statement $q \leftarrow mypat$; but the data base cannot rely on this implicit information of q . Therefore, for the function a_1 , q is nothing more than a string of characters, and its membership in $P'(U)$ must be verified explicitly.

To summarize: the implicit information carried by the outcome of a D-function into the storage space of Π , cannot be carried back into the data base. This loss of information, which may cause serious difficulties in the enforcement of certain D-rules, is due to the complete freedom that Π has in the manipulation of its own storage space. To maintain the credibility of such implicit information, it seems necessary for the data base to have some control over what happens to the outcome of a D-operation, even when it is stored in the storage space of Π . For that one must first be able to impose

restrictions over what may happen inside a program, or rather, over what a program may do to its own information space.

Such a capability is indeed featured by most high level programming languages, it may be called the restrictability of a language. For example, in some ALGOL like languages, a variable declared to be of type integer is guaranteed to contain integer number, regardless of what the program may do; in the SIMULA language, once a variable is declared to contain a pointer to an object of a given class, there is no way for the program to store anything else in this variable. This ability to restrict a program is not carried far enough in conventional programming languages, and will have to be extended for our purposes. In particular, it should be possible to impose restrictions from outside the program, from a sub-schema, say. Techniques for doing so were discussed in detail in [3]. (What we have in mind is illustrated schematically in figure 2).

Coming back to the case of example 2, its privacy rule can be enforced by imposing the following restriction on Π ; the "type" of the variable q , used as an argument for the D-functions a_1 , must be defined in such a way that only outcomes of the D-function $mypat$ can be stored in it. For details the reader is referred to [3], let us only point out that this restriction can be imposed without any run time checking.

Let us consider yet another example.

Example 3:

Let T be a binary tree in D which consists of the set of nodes $\{n\}$. Let $left(n)$ and $right(n)$ be D-functions defined over the nodes $n \neq 1$ such that $left(n)$ and $right(n)$ are the left and right sons of the node n , and the outcome of $data(n)$ is the data stored in the node n . Let $tree(n)$ be the subtree of T whose root is the node n . Let U be a user who has the right to retrieve the data stored only in the subtree $tree(n_0)$, for a given node n_0 . To enforce this privacy rule it may seem to be sufficient to provide U with an image $\bar{D}(U)$ which contains nothing but the node n_0 and the functions $left$ and $right$ and data. Using these functions the user can get to every node $n \in tree(n_0)$, moreover he seems to have no way of getting to any node outside $tree(n_0)$. Unfortunately, however, there is nothing here to prevent U from getting a pointer to a node $n' \notin tree(n_0)$ from an outside source, and to apply the function data to it.

As a second attempt to enforce our privacy rule one may try to check the argument of $data(n)$ for membership in $tree(n_0)$, whenever it is invoked. But such a check may be so extremely inefficient, that it can safely be ruled out as a viable solution to our problem.

Thus, the only way left for the enforcement of our privacy rule is again to impose restrictions of Π . The following restriction would be sufficient here: The "type" of the variable n , used as an argument of the D-functions $left$ and $right$ and data, must be defined in such a way that the content of a must

INTENTIONAL RESOLUTION OF PRIVACY PROTECTION
IN DATABASE SYSTEMS

N. Minsky

Department of Computer Science
Rutgers University
New Brunswick, New Jersey

Intentional Resolution of Privacy Protection in Database Systems

Naftaly Minsky
Rutgers University

1. Introduction

Traditionally, privacy protection in database systems is understood to be the control over what information a given user can get from a database. This paper is concerned with another, independent, dimension of privacy protection, the control over *what a user is allowed to do with a piece of information supplied to him by the database*. The ability to restrict the use of information retrieved from a database will be called "*intentional resolution of privacy protection*."

It may seem to the reader that *international resolution* is mainly a legal matter, not within the realm of computer technology, because once a piece of information is revealed to a human user, there is no way for the database system to impose restrictions on how to use it. However, the database does not usually communicate information directly to the *human user* but to a program written by him. In fact, much of the information retrieved from a database is not intended for the programmer's eyes, but for his program. For example, it can happen that confidential information is needed in the user's program as a means for getting other, sometimes less confidential data, which is what the user is really after. Were it possible to prevent the user's program from revealing to the user himself certain information given to it by the database (by printing the information itself or any computed result of it), one would be able to release to the *program* data which for privacy reasons should not be released to the programmer himself. In the absence of such *intentional resolution* capability one is forced to release either too much or too little information to the requesting program.

Consider, for example, a programmer who works for the "database administrator." Since it is the job of such a programmer to maintain the information in the database, he must be given almost universal permission to retrieve data from it. Thus, although most of these programmers have no need to actually see the information manipulated by them, they can usually do so, which according to many observers, presents the most serious threat to the privacy of databases. We would like, therefore, to be able to prevent a programmer from actually seeing certain information which is released to his program. In other words, we sometimes want the programmer to "*eat the cake blindfolded*." This is one aspect of *intentional resolution*.

Unfortunately, intentional resolution is fundamentally incompatible with the traditional approach to privacy protection, which is based primarily on access

Traditionally, privacy protection in database systems is understood to be the control over what information a given user can get from a database. This paper is concerned with another, independent, dimension of privacy protection, the control over what a user is allowed to do with a piece of information supplied to him by the database. The ability to condition the supply of information on its intended use is called here "*intentional resolution*" of privacy protection.

The practical importance of intentional resolution is demonstrated by several examples, and its realization is discussed. It is shown that intentional resolution can be achieved, but that it involves a radical change from the traditional approach to the process of user-database interaction. In particular, it appears to be necessary for the database to impose a certain amount of control over the internal behavior of users' programs which interact with it. A model for user-database interaction which admits such a control is developed.

Key Words and Phrases: protection in databases, protection in programming languages, privacy, security, intentional resolution of privacy, interaction with databases

CR Categories: 3.50, 3.70, 4.20, 4.30

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This work was supported by Grant DAHCIS-73-G6 of the Advanced Research Projects Agency. Author's address: Department of Computer Science, Hill Center, Busch Campus, Rutgers University, New Brunswick, NJ 08903.

¹ In this paper, we will use the term "privacy protection" as a *restriction on the retrieval process only*. This is admittedly a limited sense of the term, since updates of the database should also be subject to privacy restrictions. Note also that many writers are using the term "security" for what we call "privacy."

control.² That is, the "privacy enforcer" is usually viewed as a kind of mediator between a database and its "user," where by the term "user" we mean the human user together with his program (see Figure 1(a)). Such a privacy enforcer has the authority to decide which information could be transmitted to this "user complex," but it has no say on how such information is to be used, either by the human user or by his program. Indeed the privacy enforcer has no more control over the user's program than it has over the user himself.

It follows that in order to achieve any degree of *intentional resolution* it is necessary, in addition to the normal access control protection, to give the database some control over what happens inside the program which interacts with it. In particular, there should be a way to restrict the program in what it can do with the

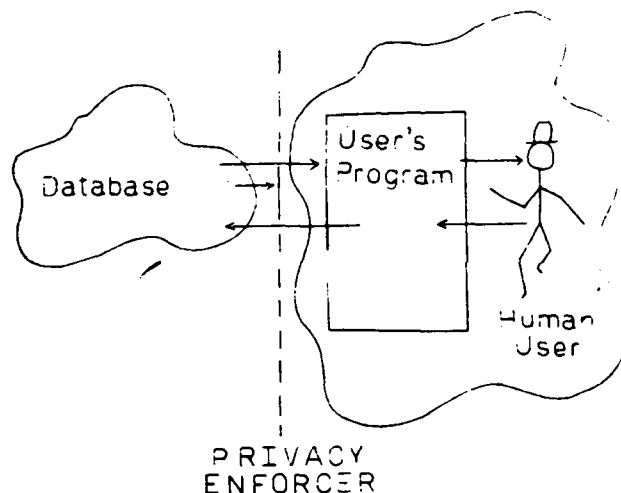
information retrieved from the database (see Figure 1(b)). As we will show in this paper, a useful level of protection can be achieved by means of only a modest degree of control over the user program, which can be imposed quite efficiently, often without any run time checking. The approach to be proposed is based on well-known techniques of protection in programming languages [11] and in operating systems [5], but it is believed to be novel in the context of databases and it calls for a radical change in the traditional approach to user database interaction.

In Section 2 we summarize the basic principles on which the conventional *access control* protection mechanisms are based. The limitations of this kind of protection are illustrated by an example. In Section 3, some relevant aspects of programming languages are discussed. Our model for user-database interaction is presented in Section 4, and its use for privacy protection is discussed in Section 5.

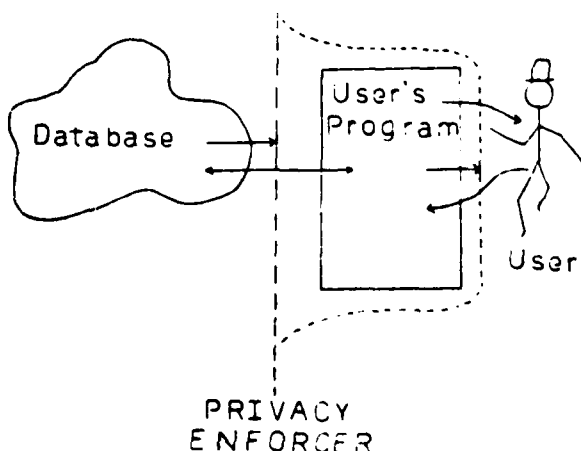
Finally, in Section 6, the scope of application of our protection technique is discussed. In particular, it is pointed out that this technique is not intended for all types of users of a database.

Fig. 1. Two approaches to privacy protection.

(a) The traditional approaches to privacy protection do not distinguish between the human user and his program.



(b) Here there is a clear distinction between the user and his program. The user's program is under (partial) control of the privacy enforcer of the database, so that the program cannot communicate to the user every piece of information it gets from the database.



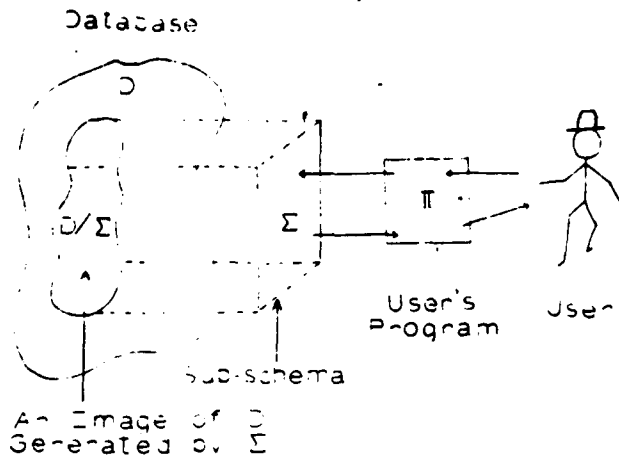
2. Access Control Protection Mechanism

We begin by presenting a schematic and simplified model for the *process of user database interaction* (see Figure 2), which we believe is consistent with the conventional approach to databases. The main participants in this interaction are the user (U) and his program (Π), on one hand; and the database (D) on the other. However, in general, the user is not presented with the database itself, but with an *image* of it, which contains only part of the database, possibly in some abstract form, [1, 2, 3]. Following the DBTG report [1], we will use the term *subschema* for the specification of such an image, denoting it by Σ . (The image of D which is formed by a specific Σ will be denoted D/Σ (see Figure 2)). There is no general agreement as to the exact nature of the subschema concept, but from the viewpoint of protection it can be described as the specification of which parts of the database a given user is allowed to access, and what operations he is allowed to apply to them. In other words, the subschema is a collection of "capabilities"³ to be granted to a certain user or to a class of users. Of course, the database would usually have many users with diverse needs and authority; it must therefore contain many subschemas to support the various necessary images. To interact with a database the user must first be *connected* to one such subschema. Assuming that the user has an unforgeable

² This is true only in the context of databases. More sophisticated protection techniques are being developed for operating systems, (see [5], for example).

³ At this point we are using the term "capability" in its colloquial sense. A more technical meaning will be given to this term later.

Fig. 2. An illustration of the conventional view of user-database interaction.



identification, this *connection* can itself be under the control of a protection mechanism which decides which subschemas can be connected to which user.

The *process* of user-database interaction can be viewed as a sequence of "operations" applied by the user program Π to the database D , which may be either retrieval requests or update operations.⁴ We will refer to any such operation as a *D-operation*.

The *access control protection mechanism* can now be defined as the validation that: (a) each and every *D-operation* issued by Π is permitted by the subschema Σ to which the user is connected; (b) information to be transmitted by D to Π , in a response to a *D-operation*, belongs to the image D/Σ .

Without going into the details of such a mechanism, the following can be stated: From the point of view of the *access control* protection, the user program (or process of computation) can be considered as a "black box" which issues *D-operations*, the internal behavior of Π being irrelevant to this kind of protection. This is true in principle, even if for efficiency reasons one decides to examine the user program in order to identify the potential *D-operations*, and check their validity at compile time. Such a technique, proposed by several writers [3, 4], can still be considered as a version of access control technique, since it is not interested in the behavior of Π itself, only in its *D-operations*. Some of the limitations of an access control protection mechanism will now be illustrated by an example.

Consider a binary tree stored in D which, in set-theoretical notation, is defined to be the set T of "nodes" together with the three functions $DATA$, $LSON$, and RSN , which are defined over T . For a given node $n \in T$, the value of the function $DATA(n)$ is considered to be the "data" associated with the node n ; the function $LSN(n)$ returns the node $n' \in T$ which is the

⁴ These may be high-level operations loaded with semantics, such as "fire an employee" (see [2]).

"left son" of n , and similarly for $RSN(n)$. Now let N be a certain node in T , and let U be a user whom we wish to grant a read-access *only* to the subtree of T whose root is N , which we will denote by $tree(N)$. How can such a restriction be imposed?

In order to make $tree(N)$ accessible to U , it is enough to provide the user with the node N , and to allow him the use of the three functions $DATA$, LSN , RSN . It might seem that with just these four items the user cannot get to any node outside of $tree(N)$. This, however, is not the case. Suppose, for example, our user manages to get a node identifier $n' \notin tree(N)$ from an outside source. Since he is allowed to use the three functions that are associated with our tree, this would give him an access to the entire subtree $tree(n')$. The only solution that an access control protection has to offer for this problem is to check the argument of $DATA(n)$ for membership in $tree(N)$ whenever this function is invoked. This brute force protection technique can become extremely inefficient for large trees, and can be considered impractical. A potentially better method is suggested by the following observation.

Let t be a "type" defined in the program Π , and suppose that we manage to impose the following rules on Π :

Rule 1. The only "things" which can be stored in a variable of type t are the node identifier N or outcomes of the functions LSN , RSN .

Rule 2. The functions $DATA$, LSN , RSN can be applied to the variables of type t only.

It is quite clear that if Π does not violate these rules, it cannot access any node which does not belong to $tree(N)$. Moreover, it should not be too difficult to impose such rules on a program, because they are quite similar to the rules associated with *types* in programming languages. For example, there are rules built into Algol which guarantee that only integer numbers can ever be stored in an integer variable, just like our two rules above which guarantee that only nodes from $tree(N)$ can be stored in t variables. Of course, one cannot expect a language to contain rules for how a specific user should treat a specific data item retrieved from a database. We suggest, however, that such rules can be *induced* into a language at the time that the user *connects* himself to a specific subschema. The ability to induce such rules into a language depends, of course, on the programming language at hand. Therefore our next step is to examine some relevant properties of programming languages.

3. Capabilities Built into a Programming Language

In this section we will try to characterize the "capabilities" which are built into a programming language. By this we mean the "operational capabilities" provided by the language to a *process of computation* generated by a program written in this language, rather

than the "syntactic capabilities" which the language provides the programmer. For example, we will be interested in questions like what kind of operations can a computing process carry out, or what can such a process do to a certain type of information object, e.g. to an integer variable. Such questions are relevant to our subject matter because when a piece of information is transmitted to a user program we would like to know what the program can do with it. In order to be able to answer such questions we will now develop a model for the "capabilities of languages." It should be pointed out, at the outset, that this model is not intended as a contribution to the theory of languages per se, but as a tool for the subsequent treatment of user database interaction. We begin by introducing some basic concepts and notation.

Let L be a language, and let Π , or $\Pi(L)$, be a *process of computation* generated by a program written in L . If we take a snapshot of Π at a certain stage of its activity, we can talk about the *information space* of Π , to be denoted by S , which is the set of all *information objects* (or simply *objects*) which are accessible to Π at this moment. By the term *object* we mean, informally speaking, a "thing" which has a *state*. The state of an object can be observed and it may, or may not, be changeable. For example, the number "17" is a nonchangeable object, while the state of a "stack" can be changed by means of a "push" or a "pop" operator.

It would be convenient to consider some of the objects in S as being *primitive objects* provided by L to every process $\Pi(L)$, such as the various literals recognized by the language. This set of objects will be denoted by S_L . The other objects in S are generated by Π itself, as we will see later.

It is customary to classify the objects in S by means of *types*. The concept of *type* would be central to our model, but since we will be interested only in some of its aspects, to which we will have a somewhat unconventional approach, we will avoid the term "type," replacing it by "brand." We assume that the *state* of every object $s \in S$ has two distinguishable parts: *brand(s)* and *value(s)*. Here, *value(s)* stands for the "main part" of the object s , and *brand(s)* is a tag which will be used to classify the various objects in S . For simplicity we assume that there is a fixed and finite set of different *brands* for a given language L , to be denoted by B_L . We will use the symbol β to denote a generic brand, and barred capital letters to denote specific brands. For example, we will use I as the brand associated with integer numbers, R as the brand associated with real numbers, T as the brand associated with text objects, etc. An object whose brand is a certain β will be referred to as a β -object.

For reasons which will not be spelled out here we introduce the following convention: the language will recognize a special "null" object which will be treated as if it is branded by all the brands in B_L .

The objects in S are generated and manipulated by

means of a set P_L of *primitive operators* provided by the language. Examples of such primitive operators are: the "+" operator which adds two numbers (objects), generating a new number which is equal to their sum; or a "declare" operator which generates a new array. (Note that we do not distinguish between operators that can act at "compile time" and those that act at "run time").

The *process* Π can now be formally defined as a sequence of *operations* of the form:

$$s_0 \leftarrow p(s_1, \dots, s_n) \quad (\text{for } n \geq 0),$$

which is the application of the operator $p \in P_L$ to the objects s_1, \dots, s_n in S . Such an operation may affect the objects s_i ; in addition it generates a new object s_0 , to be called the *outcome* of p , and which may in particular be null. (Note that this is not an "assignment statement"; a special *assignment operator* will be introduced later). We will assume, for simplicity, that once an object is generated into S , its brand cannot be changed; thus the operator p above can change at most only the *value* parts of objects s_i .

Under most languages a computing process is not free to apply any operator to any set of objects. The language usually has a set of rules built into it that specify which operators can be applied to which objects, and under which circumstances. This set of rules, to be called the *application rules* of the language, will be denoted by R_L . Languages may vary widely as to the type and form of their application rules; in particular, a language may have no such rules at all, as the following example shows.

Let L be the *machine language* of a classical von Neumann machine. The storage space S of every $\Pi(L)$ is fixed in this case, being the set of memory cells of the machine. The set of operators P_L available to Π are the machine instructions. There are practically no application rules here; Π is free to apply any operator to any cell. This does not necessarily result in complete anarchy, since the operators themselves may have their own built-in rules. For example, although there is no way to prevent the division operator from being *applied* to a zero denominator, the operator itself would usually refuse to divide by zero. Such internal rules, built into the operators, will not be discussed in this paper. We will look upon the various objects and operators as "black boxes," considering only *application rules* which are imposed by the language from the outside, so to speak. One class of such rules will be introduced later in this section.

To summarize, we define what we call "the *capabilities*" of a language L " as the four-tuple (S_L, B_L, P_L, R_L) , where

S_L is the set of *primitive objects* in L ,

* Note that this definition of "capabilities" is related, but not identical, to the meaning that this term has in the context of operating systems [5].

B_L is the set of *brands* defined in L ,

P_L is the set of *primitive operators* provided to every $\Pi(L)$,

R_L is the set of *application rules* imposed on $\Pi(L)$.

We will now introduce a class of application rules that will be useful to the rest of this paper.

3.1 Simple Application Rules

We now consider application rules which have the following general form:

$r: \beta_0 \Leftarrow (\rho, \beta_1, \dots, \beta_n)$ (for $n \geq 0$)

where $\beta_i \in B_L$ for $i = 0, 1, \dots, n$; and $\rho \in P_L$.

The rule r (r is a label, used for discussion only) serves two functions. First, the right-hand side of r serves as a *right*⁴ for $\Pi(L)$ to invoke any operation $p(s_1, \dots, s_n)$ such that $\text{brand}(s_i) = \beta_i$ for $1 \leq i \leq n$. The right-hand side of r will accordingly be called *right* and will be denoted by ρ . It is assumed that no ρ can appear more than once in R_L . The second function of the rule is to determine the *brand* of the outcome of an operation $p(s_1, \dots, s_n)$: it is to be the β_0 of the rule which contains the *right* for this operation (by the assumption above, there is at most one such rule). If β_0 is null and " \Leftarrow " does not appear in the rule, it would mean that no new object is generated into $S(\Pi)$, even if ρ does have an outcome.

These rules are *simple and restricted* in several respects; in particular: (a) the rules are independent of the values of their arguments, being defined in terms of the brands only; (b) the rules are invariant of the state of computation (for example, they do not distinguish between the compilation phase and the execution phase of Π); (c) the rules have "global validity," so that they cannot be used to model any "name scoping." Yet these "simple rules" can model a significant part of the application rules of many languages. For example, if I , R , and \bar{C} stand for the brands of *integer*, *real*, and *complex* objects, then the following are *samples* of the rules which one might have:

$r1: I \Leftarrow (+, I, I)$,
 $r2: R \Leftarrow (+, I, R)$,
 $r3: \bar{C} \Leftarrow (+, \bar{C}, \bar{C})$,
 $r4: R \Leftarrow (IM, \bar{C})$.

The rule $r1$ allows Π to add two integers, specifying that the outcome is an integer. $r2$ allows Π to add an integer to a real number (obviously there is a conversion to be done here, but this is irrelevant to the application rules). $r4$ allows Π to apply the operator IM to a complex number, which presumably returns its imaginary part. Note also that if the set R_L at hand does not contain the right $(+, I, \bar{C})$, then Π cannot add, directly, an integer to a complex number.

⁴ The term "right" is borrowed from [5].

⁵ Such a distinction is made in most languages. For example, in Fortran variables are generated only at compile time.

As another example, suppose that we have in L the data type "stack of integers" whose brand is S . Assuming the existence of the two operators $PUSH$ and POP in P_L , we might control the use of stacks by the following rules:

$r1: (PUSH, S, I)$
 $r2: I \Leftarrow (POP, S)$

Note that according to $r1$, $PUSH$ has no outcome.

In order to get a better approximation to realistic programming languages we must introduce explicitly the concepts of *variable* and *assignment operator*. This we will do next.

3.2. Variables and Assignment Operator

The concept of "variable" can be introduced into our model, intuitively, as follows: a *variable* is an object whose *value* part is another object or the "name" of such an object, if sharing is desired (we will ignore the distinction between these two cases). The *value* of a variable is assigned to it by the *assignment operator* " $:=$ ". In our notation the assignment operation will have the form $:= (\text{variable}, \text{object})$, and for simplicity we assume that this operation has no outcome. If a variable appears as an argument of an operation in any other context, then the object which is its value is assumed. An important property of a variable is its *domain*, which is the set of all objects which can possibly serve as values of the variable. The *domain* of a variable is clearly determined by the rules in R_L , as the following example shows.

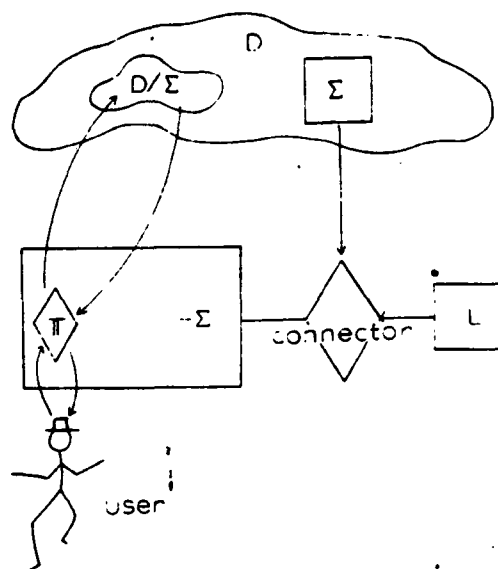
Let v and β be brands in B_L , and suppose that the following rule exists in R_L :

$r: (:=, v, \beta)$

This rule allows v -objects to appear at the "receiving end" of an assignment statement, the other end of which is a β -object. Formally, this makes v -objects into variables. In general, there may be additional rules in R_L which permit the assignment of other types of objects to v -objects. If, however, r is the only such rule, then the *domain* of a v -object, as a variable, is the set of all β -objects. We will call such a variable a β -variable; we will also refer to it as a *simple variable*. It is clear that if an object s is stored in a simple variable, then its brand does not have to be explicitly stored. Moreover if all the variables in Π are simple, then simple application rules can be enforced at compile time.

These properties are the main reason why many programming languages have only *simple variables*, e.g. *integer variables*, *real variables*, etc. In fact, simple variables are so common that one tends to forget the difference between the *type* (brand) of an object and the domain of a variable; indeed the term "type" is usually used for both, although it is clear that an "integer number," for example, and an "integer variable" are not similar objects and do not satisfy the same rules.

Fig. 3. An illustration of the proposed approach to user-database interaction. In order to interact with the database, the user must first effect a "connection" between a language L and a subschema Σ . This connection generates a "new" language L_Σ . The user's program would operate from within L_Σ , so to speak. It has, therefore, the necessary capabilities to interact with D/Σ .



In the rest of this paper we will assume, first, that the user program is written in a language whose *capabilities* can be described as above.⁸ This means, in particular, that the *application rules* of the language are strictly enforced (such languages are often called "strongly typed" languages). Second, we want these rules to be explicit, rather than implicit in the overall structure of the language. This second property will allow us to augment the application rules of the language by the rules built into a subschema of a database, as we will see next.

4. A Model for User-Database Interaction

We have already seen, in Section 2, that in order to interact with an image D/Σ of a database D , a user U must first be *connected* with the subschema Σ , which in some sense contains the "capabilities" necessary for interaction with D/Σ . Retaining this basic notion we will now give it a new interpretation. We begin with a general outline of our approach; a more detailed discussion will follow.

Let L be the programming language to be used by U for his interaction with the database. This language might have some special tools for interaction with

⁸ This does not imply that every user should use such a language, but only those who wish to take advantage of intentional resolution (see Section 6).

⁹ This language amplification is analogous to the extension of a language by new data types or new "clusters" [11], except that here the extension would be induced into the program from the database, and not defined by the programmer himself.

databases, such as the DML features of the DBTG [1], but it cannot have the specific capabilities which are necessary for the interaction with a *given* image D/Σ . Presumably, such capabilities exist only in Σ . In order to provide our user with these capabilities, we will *augment* the capabilities of the language L (in the sense of Section 3), by the capabilities of Σ . This action will be called the *connection* between L and Σ . In a sense, such connection generates a new " Σ -amplified" language, which we will denote by L_Σ . This "newly created language," which exists only for the duration of a single session of interaction between U and D , has the same syntax and most of the semantics of the "base language" L . It is an amplified language only in the sense that it contains capabilities which do not exist in L .⁹ In particular, Σ may contain new *brands* (types) and new rules which are induced into the newly formed language. Thus a program written in L_Σ will be under the joint jurisdiction of the set of rules built into L , and those in Σ , which is the effect we are trying to achieve. Figure 3 is an attempt to illustrate this model of user-database interaction. A more formal treatment of this model is presented below.

4.1. The Subschema and Its Connection to L

The subschema will be viewed here essentially as a collection of capabilities which facilitate interaction with a given image of the database, and which are designed to augment the capabilities of a given language L .¹⁰ Formally, Σ is defined to be the 4-tuple $(S_\Sigma, B_\Sigma, P_\Sigma, R_\Sigma)$ where:

S_Σ is a set of (names of) *information objects* in D , which are to be accessible to the user. They will be called *D-objects* (e.g. sets, relations, records of various types, etc). The specification of the brand β associated with an object $Q \in S_\Sigma$ will be done by the notation " $Q: \beta$ ".

B_Σ is a set of *brands*, which are all different¹¹ from the brands in B_L .

P_Σ is a set of *operators* (or procedures) which are defined in the database, and which the user is allowed to use. These operators will be called *D-operators*.

R_Σ is a set of *application rules*, to be called *D-rules*, designed to regulate the interaction of a user program with the database. We again restrict ourselves to "simple rules," which have the following general form:

$$r: \beta_0 \leftarrow (p, \beta_1, \dots, \beta_n) \text{ (for } n \geq 0)$$

where $\beta_i \in B_\Sigma \cup B_L$ for $0 \leq i \leq n$, and $p \in P_\Sigma \cup P_L$.

This is exactly the structure of the "simple rules" in

¹⁰ This does not mean that the interaction with the database must be carried out by means of a single language. Different subschemas may be designed for different languages.

¹¹ To distinguish between the brands in Σ from those in L , we will denote the former by barred lower case letters, e.g. \bar{a}, \bar{b} ; while the latter are denoted by barred upper case symbols. When we do not wish to distinguish between B_Σ and B_L we will use $\beta, \beta_1, \beta_2, \dots$.

R_L except that these rules can use the operators and brands in Σ as well as those in L .

This concludes the formal definition of the subschema. The connection of Σ with L can now be viewed as an augmentation of the capabilities of L , namely the sets S_L, B_L, P_L, R_L , with the corresponding sets $S_\Sigma, B_\Sigma, P_\Sigma, R_\Sigma$. This augmentation creates the Σ -amplified language L_Σ whose capabilities would be

$$(S_L \cup S_\Sigma, B_L \cup B_\Sigma, P_L \cup P_\Sigma, R_L \cup R_\Sigma).$$

In practice, this means that the compiler of the language L should be able to accept the new brands, rules, etc. which are defined in Σ , and it should enforce the rules $R_\Sigma \cup R_L$ on the user program¹² Π .

One of the main reasons for introducing the above model was to give the database a degree of control over what happens with information transmitted to the user's program. Here is how it works. Let T be a D -operator, and let $r: \beta \leftarrow (T, \dots)$ be a rule in R_Σ . Let $t \leftarrow T(\dots)$ be an operation authorized by r . The object t generated by T (which may simply be a piece of data retrieved by T from the database) would, by rule r , be branded by β . To see what the future of t might be, we will distinguish between two cases, depending on β .

First, if β is a member of B_L then there would be rules in R_L which specify how such objects can be manipulated. For example, if β is I then this object can be manipulated like any other integer number. In general, an outcome of a D -operation which is branded by one of the standard brands of B_L is released from the control of the subschema and comes under the jurisdiction of the language L .

Suppose now that β is a member of B_Σ . In this case there is no rule in R_L which authorizes any operation on t . The only operations which are applicable to t are those which are authorized by rules in R_Σ . Thus although the object t resides in the storage space of the user's program, it is still under the jurisdiction of the database. For example, the program would not be able to multiply, compare, or print the object t unless such operations are permitted by R_Σ .

Before considering an example, we will introduce some conventions and notation which will simplify the subsequent discussion.

(a) For every brand β explicitly included in B_Σ , we assume that there is a companion brand in B_L which is the brand of β -variables (see Section 3.2). That is, a program written in L_Σ should be able to generate (or declare) β -variables for every brand β explicitly included in B_Σ in much the same way that "integer variables" are declared in Algol.

(b) Because it is sometimes necessary to generate a copy of an object, assigning a different brand to it, we

¹² Whenever there is no danger of confusion we will not distinguish between the user's program and the process of computation generated by it; both will be denoted by Π .

introduce a special operator, called *CONVERT*, which does just that. For example, if we have the rule $\beta_2 \leftarrow (\text{CONVERT}, \beta_1)$ it means that the operation $s_2 \leftarrow \text{CONVERT}(s_1)$ can be invoked for any β_1 -object s_1 , generating an object s_2 such that $\text{brand}(s_2) = \beta_2$ and $\text{value}(s_2) = \text{value}(s_1)$. The operator *CONVERT* is assumed to be included in P_L .

(c) We assume that there is a universal retrieval operator *GET*, which retrieves an arbitrary member of any set in the database. *GET* will be assumed to exist in every P_Σ , but its application to a given set, and the brand of its outcome, must be explicitly specified by rules in R_Σ . (Similar assumptions are made concerning a universal storage operator *PUT*.)

4.2. An Example

To clarify all this, let us return to the binary tree example of Section 2. Consider the following subschema:

$$\begin{aligned}\Sigma &= \{S_\Sigma, B_\Sigma, P_\Sigma, R_\Sigma\}, \\ S_\Sigma &= \{N: \bar{n}\}, \quad B_\Sigma = \{\bar{n}\}, \\ P_\Sigma &= \{LSON, RSON, DATA\}, \\ R_\Sigma &= \{r1: \bar{n} \leftarrow (LSON, \bar{n}) \\ &\quad r2: \bar{n} \leftarrow (RSON, \bar{n}) \\ &\quad r3: I \leftarrow (DATA, \bar{n})\}.\end{aligned}$$

First, note that there is a brand \bar{n} , defined in Σ , which would therefore be induced as a brand (or type, if you will) of the language L_Σ . By convention (a) this means that a program Π written in L_Σ can generate (or declare) \bar{n} -variables, namely, variables which can contain only \bar{n} -objects. We need to ask how \bar{n} -objects can be generated, and how they can be manipulated.

One \bar{n} -object is the node N which is declared in S_Σ to be \bar{n} -branded. In addition, the subschema provides Π with the operators *LSON* and *RSON*, which, by rules $r1$ and $r2$, can generate \bar{n} -objects. Note that these are the *only* ways to generate \bar{n} -objects. Even if Π manages to get a node $n' \in \text{tree}(N)$ from an outside source, it cannot make it into an \bar{n} -object. Moreover, once an \bar{n} -object is created it cannot be modified in any way; there simply is no rule which permits that.

Now, since \bar{n} -objects are the only type of objects which can be used as an argument for any of the operators *LSON*, *RSON*, and *DATA*, it is clear that an \bar{n} -object belongs to $\text{tree}(N)$. (Note the correspondence with the two rules formulated in Section 2 for this example.) Moreover, if \bar{n} -objects are stored only in \bar{n} -variables, then all these rules can be enforced at compile time.

Note that the only "printable" information that the user can get from all this is the outcome of the operator *DATA*, which is I -branded and is, therefore, a "normal integer." (It is assumed here that every node in our tree contains a single integer value as its *data* part.)

As a further illustration, let us consider a small variation of the above. Suppose that for some reason the

user is interested in examining the \bar{n} -objects themselves. We can easily permit that without compromising our security requirements. Suppose that an \bar{n} -object is actually an integer number, which would be the case if it is some sort of pointer to a node. We can add to R_2 the rule

$r4: I \leftarrow (CONVERT, \bar{n})$

using the standard operator *COPY* introduced by convention (b) above. In effect, $r4$ allows Π to copy the value of any \bar{n} -object into an integer variable, but it does not allow Π to change the \bar{n} -object itself.

5. Techniques for Intentional Resolution in Privacy Protection

In this section we will discuss several types of protection problems, and their proposed solution. What is common to all these problems is that they cannot be solved by access control alone; or, at best, their access control solution is very difficult.

5.1. Preventing Undesirable Analysis of Data

The power of the database as a source of information is due not only to the fact that there is a great deal of data stored in it which can be retrieved, but also to the ability to correlate various pieces of data with each other. This ability, which sometimes carries very sensitive information, can be denied to a user, as is illustrated by the following example.

Let D be a database maintained by a *credit bureau*, and let $E = \{e\}$ be a set of records about financial transactions. Let the structure of such a record be $e = (\text{pname}, \text{trans}, \text{company})$ where *pname* is the name of the person involved; *trans* is the description of the transaction, and *company* is the name of the company involved. The main security problem in such databases is, of course, the individual privacy of the people; However, here we will consider this problem solved and deal with the *privacy of the companies involved*.

The problem is this: although the credit database is intended to provide information about people, it can be used for the analysis of the financial state of a company. Such analysis can usually be performed by any program which has access to the entire set E , simply by collecting all available information about a given company. To prevent a user from doing this kind of analysis we will connect him to the following subschema:

$\Sigma = (S_2, B_2, P_2, R_2),$
 $S_2 = \{E: \bar{e}\}, \quad B_2 = \{\bar{e}, r, c, \bar{j}\},$
 $P_2 = \{PNAME, TRANS, COMPANY, GET\},$
 $R_2 = \{r1: r \leftarrow (GET, \bar{e})$
 $r2: T \leftarrow (PNAME, r)$
 $r3: T \leftarrow (TRANS, r)$
 $r4: c \leftarrow (COMPANY, \bar{r})$
 $r5: (PRINT, c, \bar{j})\}.$

Rule $r1$ in R_2 is intended to mean that every record retrieved from E would be r -branded (*GET*, being a general-purpose retrieval operator.) By rules $r2, r3$, and $r4$, the three given D -operators can be applied to such a record, returning its three components. The outcome of the first two operators is T -branded; namely, it would be treated as a normal *text* and can be freely manipulated by Π . However, by rule $r4$, the *company* component would be c -branded and therefore is not released from the control of Σ . Rule $r5$ is intended to mean that c -branded data can be printed on \bar{j} -branded files, which would presumably contain the desired output of the program, and would be routed to appropriate officials. Thus the names of the companies involved can be printed on the final document, but they cannot serve as a basis for any analysis by the program which produces this document.

5.2. Hiding Intermediate Results of Computations

Consider a user who is allowed to get certain statistical information from a database, but is not allowed to see the raw data. Suppose that the statistical information can be produced by a procedure built into the database, assuming that this procedure does not reveal any confidential information to its users. The question is, how do we prevent the user from printing out the raw data which he feeds to the statistical procedure, particularly if the user has to select the data himself. This is an example of a more general situation where the user is allowed to get the final result of a sequence of computations performed in the context of a database, but he is not allowed to see the raw data or the intermediate results. In order to show that access control techniques may not be sufficient for such a situation we will analyze in detail a sequence of related privacy measures, of increasing complexity, for a fairly general class of problems.

Let E_0 be a set of records in the database D , and let T_1, \dots, T_k be D -operators. For a given $e_0 \in E_0$, let e_1, \dots, e_k be defined by the following equations:

$$e_1 = T_1 e_0; e_2 = T_2 e_1; \dots; e_k = T_k e_{k-1}$$

and let E_i be the set of all such e_i for $1 \leq i \leq k$.

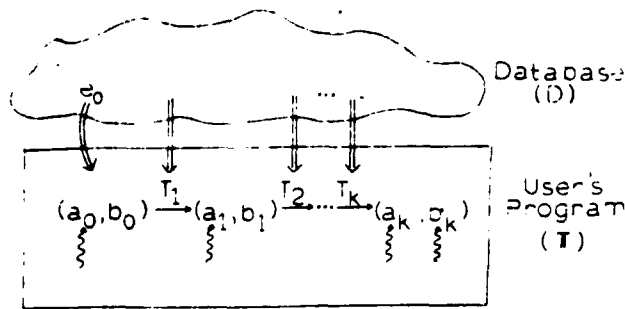
First, suppose that U is allowed to see E_k but is not allowed to see the intermediate results E_i ($i < k$) or the raw data E_0 . This privacy measure can easily be enforced by means of standard access control techniques, provided that U can define E_0 to the database without actually examining it. All we have to do is to permit the user to apply the composite operator

$$T = T_k \circ T_{k-1} \circ \dots \circ T_1$$

to E_0 .

Suppose next that every one of the records e_i is a pair: $e_i = (a_i, b_i)$, and that the user is allowed to examine a_i , but not b_i , for $i < k$. (The situation is illustrated in Figure 4, where the wavy arrows indicate which information should be released to Π , and double

Fig. 4. Hiding intermediate results.



arrows indicate information, including operators, derived directly from the database.)

The situation now is more complicated because the user might want to use his right to inspect a_i in order to select members of E_0 and to decide whether or not to continue with a particular sequence of transformations. It is not possible anymore to replace the individual transformations T_i by the composite transformation T , because the latter does not provide for any user intervention. The user's program itself should be able to generate the partially confidential records e_i . Moreover, an arbitrary number of e_i 's may be generated and maintained by Π at any given time. Since these records are not part of D proper, it seems that they belong to the storage space of Π . Therefore to maintain their secrecy Π must be restricted as to what it can do with these records. And yet, the situation can still be handled by means of access control, even if very inefficiently, as follows:

Suppose that, when a record e_i is generated by T_i it is stored somewhere in the database (although it does not logically belong there). Only the nonsecret component a_i of e_i is actually transmitted to Π , together with a pointer to e_i , to be denoted by p_i . When Π applies the operator T_{i+1} to the pair (a_i, p_i) , the database will use p_i to locate the complete record e_i . Since the pointer p_i does not carry any meaningful information for the user, it can be safely released to his program. Thus, although it would be very difficult and inefficient, it is still possible (at least in principle) to enforce the privacy restrictions at hand with access control alone.

As a final version of our privacy measures we will now consider the following situation: Suppose that although the value of the components b_i of e_i should not be released to the user, he is allowed to manipulate these components in various ways. For example, the user's program might be allowed to print the entire record e_i , including b_i , on a certain file; (assuming that this file would be physically unavailable to the user). Or he might be allowed to compare the various b_i 's with each other, but not with anything else. (This capability might reveal to the user certain limited amounts of information about b_i .) Now, since Π is allowed to perform certain "normal" operations with

the records e_i using the primitive operators of the language L , it is clear that e_i must be stored in the storage space of Π . Since the secrecy of the component b_i of e_i must still be protected, we must be able to impose restrictions on Π . Thus *access control is, in principle, not powerful enough for such a case*. The way to handle this problem by means of our techniques is left for the reader.

5.3. The Need for Local Confinement

We now will consider a class of problems for which the protection techniques of Section 4 are not sufficient and would have to be refined. We will begin with a very simple example.

Consider a database D and a highly confidential set of records F in it. Let U be a programmer who is commissioned to program and apply a transformation T to every record $f \in F$ and to store the transformed records back into the database as a set F' . Suppose that due to the confidentiality of F we do not want its content to be revealed to the programmer U himself, or to be leaked in any form or shape to the outside world. (This is an example of the class of problems mentioned in the Introduction. As was pointed out there, the solution of this kind of problem has great practical importance.)

The difficulty with this case is that since our user has to program the transformation T himself, the confidential records $f \in F$ must be *released* to his program so that they can be examined and manipulated. This means, however, that the records retrieved from F would not be under the jurisdiction of the D -rules, and we are back where we started.

One solution to our problem is to *confine* the user's program, in the sense of Lampson [7], namely, to block all the "output channels" of Π so that it cannot communicate to the outside world any information about F while it is still allowed to write into F' . However, such a complete confinement of the user's program may not be desirable because the same program which manipulates the confidential information may also be doing other things which might require various output channels. Thus instead of the confinement of the entire program Π , we would prefer a "local confinement" only of those parts of Π which need a free access to F . For this we introduce a new linguistic construct to be called the *confidence module*.

5.3.1. Confidence modules. Let M be a *module*¹³ of a program Π , such as an Algol block or procedure. The storage space of Π can be divided into two mutually exclusive parts with respect to M : the *internal storage space* of M , which is accessible only from within M , and the *environment* of M , to be denoted by $E(M)$, which is all the rest. $E(M)$ would include not only variables defined within Π , but also files and those parts

¹³ The concept of "module," as well as some other standard concepts to be mentioned below, will not be defined here.

of the database which are accessible to II. Normally a module has a variety of means by which it can affect its environment and thus pass information to it. For example, M may change a variable which is in its scope but is not internal to it; it may write on a file, etc. Informally we will refer to all such means that M has to affect its environment, as the *output channels* of M . (Note that even the existence of multiple exit points out of the module should be considered as an output channel.) Usually one does not have much control over which output channels a given module will have. For example, if a file is accessible to a program it is usually accessible to *all* its modules. This brings us to the following concept.

Definition. A *confidence module* is a module which has only a specified set of available output channels.

We will not say how exactly *output channels* should be specified, but an example might clarify the concept. Let F be a file name, and let \bar{b} be a brand; then

confidence(F, \bar{b}) *begin...end*

would be a module whose output channels consist of the file F and any \bar{b} -branded variable that this module can access.

Note that there are no restrictions on what a *confidence module* can "see," since we are concerned only with how a module can communicate information to its environment. Thus, for the purpose of read-only access, a confidence module may have the normal scope of an Algol block, say.

A systematic study of the concept of confidence modules and their implementation is outside the scope of this paper. The interested reader is advised to study Lampson's paper [7], which discusses the problems involved in the confinement of programs. (It should be pointed out, however, that Lampson's paper is concerned primarily with the operating system aspects of confinement. Here, on the other hand, since we are talking about a linguistic construct, the *compiler* should carry the main responsibility for enforcing confinement.)

5.3.2. Use of confidence modules. In order to use the confidence modules for intentional resolution we will have to generalize our *application rules*. As was already pointed out, the scope of a *simple rule*, introduced in Section 3, is the entire program II. That is, operations which are legalized by a certain rule r can be carried out anywhere within II. We now introduce a restriction of the scope of an application rule to a certain type of confidence modules, as follows:

Let r be an application rule; then the rule

$r' : r / \text{confidence}(c_1, \dots, c_n)$

is identical to r but is valid only within confidence modules which do not have any output channels except c_1, \dots, c_n . (It is assumed here that c_i denote "output channels".) The usefulness of this generalization comes

simply from the fact that *one might be willing to let the user program see more of the database, but within a confidence module from which he cannot communicate freely*. This is readily demonstrated by the following treatment of the example which was given at the beginning of Section 5.3.

Consider the subschema defined by:

$\Sigma = \{S_z, B_z, P_z, R_z\},$
 $S_z = \{F: \bar{f}, F': \bar{f}'\}, \quad B_z = \{\bar{b}_1, \bar{b}_2, \bar{f}, \bar{f}'\},$
 $P_z = \{GET, PUT\},$
 $R_z = \{r1: \bar{b}_1 \Leftarrow (GET, \bar{f})$
 $\quad r2: 1 \Leftarrow (CONVERT, \bar{b}_1) \text{ confidence}(\bar{b}_2)$
 $\quad r3: \bar{b}_2 \Leftarrow (CONVERT, 1)$
 $\quad r4: (PUT, \bar{b}_2, \bar{f}')\}.$

The program II written in L would be allowed, by rule $r1$, to invoke the D -function $GET(F)$, which is assumed to produce a record $f \in F$ as its outcome; every such record would be \bar{b}_1 -branded. The brand \bar{b}_1 appears only in one more rule, in $r2$, which permits the conversion of a \bar{b}_1 -object into an integer, where for simplicity it is assumed that F is a set of integer numbers. Note, however, that $r2$ is applicable only within confidence modules whose only output channels are \bar{b}_2 -branded variables. Outside such a confidence module¹⁴ the records $f \in F$ cannot be manipulated or examined. Let M be such a module (see Figure 5). Since M is able to transform any f to an integer, it is possible to program an arbitrary transformation $T(f)$ in M . At the same time, M , being a confidence module, cannot affect its environment except by writing into \bar{b}_2 -branded variables. By $r3$, any integer number can be converted into a \bar{b}_2 -object. A \bar{b}_2 -branded object, in turn, can only be written in F' , according to $r4$. No other operation can be applied to it.

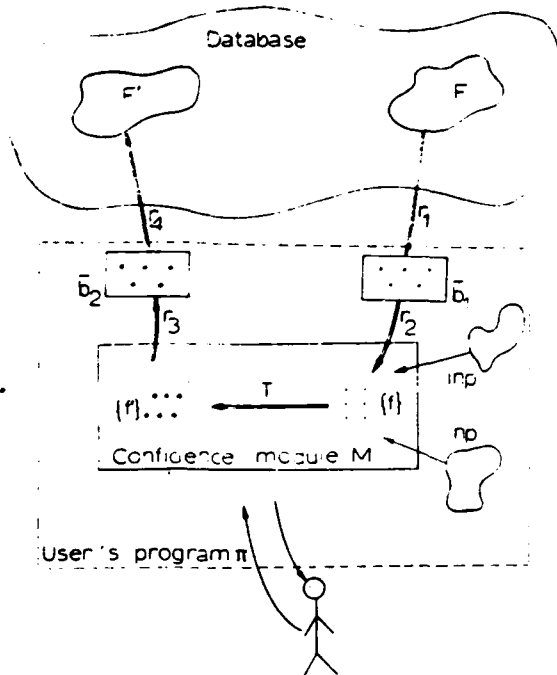
It is clear that our privacy requirements are fully satisfied. The programmer can carry out an arbitrary transformation on F , storing the results of his transformation in F' , but he has absolutely no way to see the file F' itself or any information about it, or to leak such information to anybody else. At the same time the program as a whole is not confined or limited in any sense. (The reader may wonder why the brand \bar{b}_2 is necessary; is it not simpler for the module M itself to write f' on F' ? This is indeed the case for the problem at hand. However, \bar{b}_2 would be necessary in the slightly more general case where the user is allowed to apply certain D -operators to f' , which he might want to do outside of M .)

6. Scope of Application of Proposed Technique

A major disadvantage of the method for user-database interaction introduced in Section 4 is the strong requirements imposed by it on the language in which

¹⁴ Note that the user can have any number of confidence modules in his program.

Fig. 5. The situation created by the subschema Σ described in Section 5.3.2. The records $f \in F$ which are retrieved by Π are enclosed in an opaque box b_i which shields them from Π . (The box represents the brand b_i). The only way out of this box leads into a confidence module, which in turn has only one output channel which leads into another confinement, the box b_j . Within the confidence module M , the records $f \in F$ can be manipulated freely so that an arbitrary transformation T can be performed on them. The single arrows labeled inp illustrate the fact that there is no restriction on input channels to a confidence module. The labels r_1, \dots, r_4 on the double arrows represent the D -rule which legalizes the transmission of information represented by the arrow.



the user program is written. For example, neither Cobol nor Fortran can be used for this kind of interaction, and certainly not an assembly language. An appropriate "strongly typed" language can and should be designed, but it would be unrealistic to assume that the average user would be willing to give up his favorite language in order to interact with a database. Indeed, this is not our intention. Our method for user-database interaction is intended for the hopefully few cases in which *intentional resolution* is necessary. In this section we will try to identify some of these cases.

First, it should be pointed out that the term "user-database interaction" does not reflect correctly the way in which databases are used in general. In practice, the entity which interacts directly with a database is often some general purpose "application program," which in turn serves a number of "end-users." In such a case it may be sufficient to control the interaction between the application program and the database so that the program cannot reveal to its user any confidential information, even if it can get it from the database. If it is not necessary to control the interaction between the end-users and the application program, then the end-users can use an arbitrary programming language. The

only requirement would be that the *application program* is written in a language of the type of Section 3. Since such an application program is typically designed as an integral part of the information management system at hand, this does not seem to be an unreasonable requirement. It is quite common for an installation to impose restrictions on the language and style of programming which is used for its major systems.

There is another class of programs whose interaction with the database should be controlled by the techniques introduced in this paper. These are the *procedures built into the database itself*. Indeed it is a well-known principle of design of large systems that the interaction between every module of the system and the rest of it should be carefully controlled. We believe that this control should include intentional resolution. In this case intentional resolution can be achieved if the entire database is programmed in a single strongly typed language. This aspect of database design is discussed in [8].

Finally, it should be pointed out that the protection technique proposed in this paper is applicable not only for intentional resolution. It was shown in [10] that several other protection problems can be solved more efficiently by imposing control over the user programs than just by means of access control. Our binary tree example (Section 4) is one such problem.

7. Conclusion

The main thrust of this paper has been the introduction of *intentional resolution* as an independent dimension of privacy protection, which is complimentary to *access control*. This aspect of privacy protection is either totally ignored in the database literature, or at best it is listed as one of the unsolved problems [9], leaving a wide gap in our ability to protect the privacy of data. In particular, under most database systems the programmers who maintain an information system have a virtually unrestricted access to all the information in it. This paper is intended to suggest an approach for dealing with these kinds of protection problems, but it does not present a complete solution. This work should be completed and generalized in a number of directions; the following two are particularly important:

- The "simple application rules" are much too simple and should be generalized.
- The problem of how subschemas are generated and maintained in the database was not addressed at all in this paper. This problem can be treated only in the context of a comprehensive study of the structure of database systems and is therefore beyond the scope of this paper.

Received June 1974; revised March 1975

References

1. CODASYL Data Base Task Group (DBTG) Report, April 1971. (Available from ACM.)
2. Minsky, N. On interaction with data bases. *Proc. ACM SIGFIDET Workshop on Data Description, Access and Control*, 1974, pp. 51-62.
3. Summers, R.C., Coleman, C.D., and Fernandez, E.B. A programming language approach to secure data base access. *IBM Tech. Rep. G320-2662*, May 1974.
4. Conway, R., Maxwell, W. and Morgan, H. On the implementation of security measures in information systems. *Comm. ACM* 15, 4 (April 1972), 211-220.
5. Wulf, W.A., et al. HYDRA: the kernel of a multiprocessor operating system. *Comm. ACM* 17, 6 (June 1974), 337-345.
6. Codd, E.F. Relational completeness of data base sublanguage.

7. Lampson, B.W. "A note on the confinement problem". *Comm. ACM* 16, 10 (Oct. 1973), 613-615.
8. Minsky, N. Another look at data bases. *FDT Bull. (ACM Newsletter)* 6, 4(1974), 9-17.
9. Owens, R.C. Jr. Evaluation of access authorization for on line data management systems. *ACM SIGFIDET Workshop*, Nov. 1971, pp. 263-278.
10. Minsky, N. Protection of data bases, and the process of user-DB interaction. *Tech. Rep. SOSAP-TR-11*, Computer Sci. Dep., Rutgers U., 1974.
11. Liskov, B., and Zilles, S. Programming with abstract data types. *SIGPLAN Notices (ACM Newsletter)* 9, 4 (April 1974) 50-59.

PROFESSIONAL ACTIVITIES

Calendar of Events

This calendar aims to list scientific meetings that are open to the computing public and that are held on a not-for-profit basis. Educational seminars, institutes, and intensive courses are not included. Submittals should be substantiated with name of the sponsoring organization, fee schedule, and chairman's name and full address.

One telephone number contact for those interested in attending a meeting will be given when a number is specified for this purpose in the news release text or in a direct communication to this periodical.

All requests for ACM sponsorship or cooperation should be addressed to Chairman, Conferences and Symposia Committee, Dr. W.S. Dorsey, Dept. 503/504 Rockwell International Corporation, Anaheim, CA 92803. For European events, a copy of the request should also be sent to the European Regional Representative. Technical Meeting Request Forms for this purpose can be obtained from ACM Headquarters or from the European Regional Representative. Lead time should include 2 months (3 months if for Europe) for processing of the request, plus the necessary months (minimum 2) for any publicity to appear in Communications.

Events for which ACM or a subunit of ACM is a sponsor or collaborator are indicated by ■. Dates precede titles.

In this issue the calendar is given to January 1977. New listings are given first, and they are not repeated in the second part.

NEW LISTINGS

7-9 April 1976
Second Annual Government-Industry ADP Conference: Federal Government Data Systems, 1976-1986, Sheraton Beltway Convention Center, Washington, D.C. Sponsors: AIEE in coop. with Federal Departments and Independent Agencies in the Washington, D.C. area. Contact: Dept. PR, AIEE Seminars, P.O. Box 25116, Los Angeles, CA 90025; 213 826-7572.

17-19 May 1976
Canadian Computer Conference - Session 76, Queen Elizabeth Hotel, Montreal, Canada. Sponsors: CIPS and CSA. Contact: J.H.M. Williams, Canadian National Management Services, Box 8100, Montreal, Quebec, Canada.

18-19 October 1976
Fifth Texas Conference on Computing Systems, Thompson Conference Center, Austin, Texas. Sponsors: The University of Texas in coop. with ACM and IEEE-CS. Contact: James Snodgrass, Dep. of Computer Sciences, Painter Hall 3.28, The University of Texas, Austin, TX 78712.

29 November-1 December 1976
Bicentennial Conference on Mathematical Programming, NBS, Gaithersburg, Md. Sponsors: ACM SIGMAP and NBS Applied Mathematics Division. Conf. chm: Harvey J. Greenberg, VPI and SU, 11440 Isaac Newton Square, Reston, VA 22090.

6-8 December 1976
Winter Simulation Conference, NBS, Gaithersburg, Maryland. Sponsors: ACM SIGSIM and NBS. Chm: Harold Highland, Computer Science Department, New York State Technical College, Farmingdale, NY 11735; 516 420-2190.

PREVIOUS LISTINGS

17-19 March 1976
Ninth Annual Simulation Symposium, Tampa, Fla. Sponsor: Society for Computer Simulation (SCS) with the cooperation of ACM and IEEE-CS. Contact: L. Ed Gess, Director, Corporate Planning, Green Giant Company, Hazeltine Gates, Chaska, MN 55555; 612 448-2828.

22-24 March 1976
Conference on Data: Abstraction, Definition, and Structure, Salt Lake City, Utah. Sponsors: ACM SIGPLAN and SIGMOD. Chm: Elliott I. Organick, Department of Computer Science, Room 3160 Merrill Engineering Building, Salt Lake City, UT 84112.

29-31 March 1976
International Symposium on Computer Modeling, Measurement, and Evaluation, Cambridge, Mass. Sponsors: ACM SIGMETRICS and IFIP Working Group 7.3 (Computer System Modeling). Co-chm: J. P. Buzen, Aiken Computation Lab., Harvard University, Cambridge, MA 02138; and Arnold Ockene, IBM World Trade E/ME/A Corp., One North Broadway (WPPN-5; Dept. 465), White Plains, NY 10601.

30 March 1976
Fortran Forum III, Washington, D.C. Sponsor: ACM SIGPLAN. Chm: Frank Engel Jr., 179 Lewis Road, Belmont, MA 02175; 617 484-5911.

31 March-2 April 1976
Conference on Information Sciences and Systems, The Johns Hopkins University, Baltimore, Md. Contact: G.C.L. Meyer or W.J. Rugh, 1976 Ct., Electrical Engineering Dept., The Johns Hopkins University, Baltimore, MD 21218.

31 March-2 April 1976
ORSA/TIMS 1976 Joint National Meeting, Sheraton Hotel, Philadelphia, Pa. Contact: J.J. Burbridge, Publicity Chairman, 1976 Joint ORSA/TIMS Meeting, College of Engineering, Rutgers University, New Brunswick, NJ 08903.

1-2 April 1976
Computer Science and Statistics: Ninth Annual Symposium on the Interface, Harvard University, Cambridge, Mass. Sponsors: ASA and Harvard U. in coop. with ACM. Contact: David C. Hoaglin, Dep. of Statistics, Harvard University, 1 Oxford St., Cambridge, MA 02138.

1-2 April 1976
Third ICASE Conference on Scientific Computing: Computer Science and Scientific Computing, Quality Inn-Fort Magruder, Williamsburg, VA. Sponsor: ICASE in cooperation with ACM, ACS, AIAA, ASCE, IEEE, SIAM. Prog. chm: James M. Ortega, ICASE. Contact: Robert G. Voigt, ICASE, MS-132-C, NASA Langley Research Center, Hampton, VA 23665; 804 827-2513.

7-9 April 1976
Symposium on Recent Results and New Directions in Algorithms and Complexity, Carnegie-Mellon University, Pittsburgh, Pa. Contact: J.F. Traub, Computer Science Dep., Carnegie-Mellon U., Pittsburgh, PA 15213.

8-9 April 1976
Computer Management Symposium, Marriott Hotel, St. Louis, Mo. Sponsor: ACM SIG-UCC. Contact: Ralph E. Lee, University of Missouri-Rolla, Computer Center, Rolla, MO 65401; 314 341-4841.

13-15 April 1976
2nd International Symposium on Programming, Paris, France. Organized by The Institut de Programmation, Sponsors: Centre National de la Recherche (CNRS) and Université Pierre et Marie Curie. Contact: Secrétariat du Colloque, Institut de Programmation, 4, Place Jussieu, 75230 Paris Cedex 05, France; Tel. 325.12.21 x53.97.

20-22 April 1976
MRI Symposium on Computer Software Engineering, Barbizon Plaza Hotel, New York, N.Y. Sponsors: Polytechnic Institute of New York, Air Force Office of Scientific Research,

Army Research Office, and Office of Naval Research in coop. with ACM, IEEE-CS, and IEEE Professional Group on Reliability. Contact: Jerome Fox, Polytechnic Institute of New York, MRI Symposium Committee, 333 Jay Street, Brooklyn, NY 11201.

20-23 April 1976
Third European Meeting on Cybernetics and Systems Research (EMCSR 76), Vienna, Austria. Sponsor: Austrian Society for Cybernetic Studies. Chm: Robert Trapp, Österreichische Studiengesellschaft für Kybernetik, Schottengasse 3, A-1010 Wien 1, Austria.

22-23 April 1976
Conference on Computer Law: The State of the Art, Mark Hopkins Hotel, San Francisco, Calif. Sponsor: Computer Law Association. Contact: CLA, c/o E.J. Grenier Jr., Suite 800, 1666 K Street, Washington, DC 20006.

22-24 April 1976
Annual ACM Southeast Region Conference, University of Alabama in Birmingham, Alabama. Sponsor: ACM Southeast Region. Chm: M.P. White, University of Alabama, Box 88, University Sta., Birmingham, AL 35294.

23-24 April 1976
Symposium on Automatic Computation and Control, Milwaukee, Wis. Sponsors: IEEE Milwaukee Section, IEEE Systems, Man and Cybernetics Society, and University of Wisconsin-Milwaukee. Chm: D.D. Woelz. Contact: J.T. Snedeker, Engineering Dept., UW-Extension, 929 North Sixth St., Milwaukee, WI 53203; 414 224-4193.

26-27 April 1976
Eighth Annual Southeastern Symposium on System Theory, The University of Tennessee, Knoxville, Tenn. Sponsors: The University of Tennessee and IEEE-CS. Gen. chm: Walter L. Green, Dep. of EE, The University of Tennessee, Knoxville, TN 37916.

26-27 April 1976
Symposium on Graphic Languages, Miami, Fla. Sponsors: ACM SIGPLAN and ACM SIGGRAPH in coop. with Florida International University. Prog. Chm: Toby Berk, Dep. of Mathematical Sciences, Florida International University, Miami, FL 33144.

26-27 April 1976
Seventh Annual Pittsburgh Modeling and Simulation Conference, University of Pittsburgh, Pittsburgh, Penn. Sponsor: U. of Pittsburgh School of Engineering in coop. with the Pittsburgh Sections of IEEE and ISA. Co-chm: William G. Vogt and Martin H. Mickley, 231 Benedum Engineering Hall, University of Pittsburgh, Pittsburgh, PA 15261.

3-5 May 1976
Eighth Annual ACM Symposium on Theory of Computing, Hershey, Penn. Sponsors: ACM SIGACT and Penn State University. Prog. chm: A.K. Chandra, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

3-7 May 1976
14th Annual Association for Educational Data Systems National Convention, Adams Hotel, Phoenix, Ariz. Contact: Rick Meyer, Convention Coordinator, Phoenix Union High School District, 2526 W. Osborn Road, Phoenix, AZ 85017.

25-28 May 1976
1976 International Symposium on Multiple-Valued Logic, Utah State University, Logan, Utah. Co-sponsors: ACM, Utah State University, ONR, and IEEE-CS. Contact: Stephen Y.H. Su, Dep. of Electrical Engineering, Utah State U., Logan, UT 84321.

27 May 1976
Symposium on Trends and Applications 1976: MICRO and MINI Systems, NBS, Gaithersburg, Md. Sponsors: IEEE and NBS. Chm: M.V. Zelkowitz, U. of Maryland, 20740; 301 454-4251.

Calendar continued on page 161.

SOSAP-TR-17

March 1976

FILES WITH SEMANTICS

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAH15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

FILES WITH SEMANTICS

N. Minsky

Rutgers University
Hill Center, Busch Campus
New Brunswick, New Jersey 08903

The conventional concept of file is reexamined, and found to be unsatisfactory, both as a linguistic concept in a programming language and as a tool for data processing. A new file concept is proposed which unlike the conventional file attempts to simulate an intelligent archivist rather than a filing cabinet.

INTRODUCTION

The concept of file has an important role to play in two branches of computing: Data processing and programming language. Unfortunately, both roles are poorly performed. The conventional file, as it is provided by most programming languages and supported by most operating systems, is a (software) storage device which can be characterized by the following two properties:

1. It handles an unbounded number of records without any explicit relations between them (by the term "explicit relation" we mean a relation which is recognized and handled by the (software) device).
2. It has no "semantic content". That is to say, as far as the conventional file is concerned, the data stored in it is devoid of any meaning; the interpretation of this data is entirely up to its users.

One can draw a close analogy between such a device and a filing-cabinet (which is the source of the name "file"). However a filing cabinet is not what we need in data processing. If one has a large amount of valuable and sensitive information to be shared by a number of people, one would rather have an intelligent archivist handle this data than dump it into a filing cabinet. An archivist who would have some general knowledge about the records in his custody, should be able to validate the various transactions, thus protecting the semantic integrity and the confidentiality of the data; monitor the flow of data, reporting various unusual events; etc.

The importance of such capabilities is widely recognized, but they are mostly considered ⁽¹⁾ in the context of database systems. However, since database systems are designed to handle large volumes of complex and interrelated data, they tend to be very large and expensive. Therefore, the scope of application of these systems is somewhat limited. We maintain, however, that one needs the "intelligent archivist capabilities" for the small business with its mini-computers and few files, as well as for the big integrated corporate databases. Moreover, one should be able to get such capabilities, in this simple environment, without paying the price of a mammoth database system.

One can, of course, design a mini-database system which handles only simple data. An example of such a system is ASAP [1]. It is a well-designed system which satisfies many of our requirements. However, the scope of applications of ASAP is limited by the fact that it is a stand-alone system to be accessed primarily by a special purpose non-procedural language. Non-procedural query languages are very useful for large integrated database systems, but less so for the more modest file oriented applications. The reason is that simple collections of data are usually accessed from within a program in the context of some computation, rather than in a query type of setup. Therefore, one has to have a convenient way for interaction with such data by means of one's favorite general purpose language.

This brings us to the role that the concept of file plays in programming languages. We will find, in section 2, that the conventional file as it appears in most programming languages is most unsatisfactory, even from the viewpoint of the host language itself.

The objective of this paper is to come up with a new concept of a file which is consistent with modern trends in programming languages and which behaves more like an archivist than like a filing cabinet. Examples are drawn from an implementation of such a file concept under the SIMILA language.

(1) Unfortunately, due to the complexity of database systems, many of what we call "intelligent archivist capabilities" are rarely implemented, in spite of the widespread recognition of their importance.

(2) This work was partially supported by grant DAHCIS-73-G6 of the Advanced Research Project Agency.

LINGUISTIC VIEW OF FILES

The concept of file never enjoyed much popularity among programming language theorists. This was true at the very beginning, as the case of ALGOL [2] demonstrates, and it is still true today. Even the most recent trends in programming languages tend to leave the file in essentially the same form it had some 20 years ago. As a result of this neglect most modern languages do not answer satisfactorily to the current needs of data processing. Moreover as we intend to show in this section, the conventional file becomes a liability to the language itself.

We will restrict our discussion to one-language files, that is, to files which are accessed only by means of programs written in one given language. This excludes all sorts of message transmissions, such as a file which is to be printed out. Also excluded are files which serve as communication media between programs written in different languages.

Since a file is essentially a storage device, it is reasonable to expect it to have a total recall of everything stored in it. This, unfortunately, is not the case under most programming languages. A data item which is directly accessible to a computing process usually carries two types of information: The value and the semantics of the item. The latter includes the type of the item (e.g., "integer array"), and possibly a description of its structure (e.g., the dimensions of the array). Such semantic information might not be stored explicitly in the item; it is nevertheless known to the language processor as long as the item is stored in the internal storage space of the process. But once an object is written on a file only the value part of the item being output is actually retained. The following quote describes this phenomenon in ALGOL-68 ([3], page 298), but it could have been written about FORTRAN, COBOL, PL/I and many other languages:

"In all these [I/O] procedures, the values are first straightened, and the straightened values are transput. Thus all information as to how the original values are divided into structures...is lost."

If so, how are we to reconstruct the original items when we want to retrieve the information from the file? Here is what our ALGOL-68 source has to say about it:

"If the values output in this way are subsequently read back into a set of structures identical to that from which they originally came, then the new set will be an exact copy of the old."

Thus, we must know the structure of the items stored on a file before reading them. This is, of course, a major inconvenience, but it is even worse than that. If the content of a file is not read into suitable structures, the result may be a bunch of meaningless "things", unrecognizable by the language. Thus, the file becomes a kind of Trojan Horse which can inject illegal values into a process, violating the basic rules of the host language. For example, the structure of many languages is designed to guarantee the specified range of variables. For instance, an integer number in an ALGOL program should not find its way into a real variable. Nevertheless, it can do so by means of a file: Just write the integer on a file and read it back into a real variable. Although this possibility is clearly contrary to the spirit of ALGOL, it may not seem to be very damaging. However, a similar effect, if it happens to a data structure with a richer semantic content, has very serious consequences, as the case of "protected objects" shows.

A protected object is a term that we will use for a novel and important concept in programming languages, which appears, for example, in some versions of SIMULA [4] and in CLU [5]⁽²⁾. A protected object, or just "object", is an information item which can be manipulated only in a predefined way. Such an object is an instance of a data type defined by a module called "class" (because it serves as a template for a class of objects). The class contains a description of a structure, together with a set of procedures defined on such a structure. Some of these procedures are labeled as the operators associated with the class, the others are called internal procedures. The objects which are instances of a class are protected in the sense that they can be manipulated only by means of the operators defined on them; the internal structure of an object, as well as its internal procedures are not visible from the outside. Consider, for example, the following class which defines stacks:

1. Even if this space is virtual.

2. We will take some liberties in describing this concept, borrowing from both SIMULA and CLU, and using our own terminology. The class concept as described here, actually exists only in the PDP/10 version of SIMULA.

```

CLASS STACK (N); INTEGER N;
BEGIN
  STRUCTURE: INTEGER ARRAY A[1:N];
             INTEGER TOP;
  OPERATORS:
    PROCEDURE PUSH (V); INTEGER V;
    IF TOP=N THEN (TOP:=TOP+1; A(TOP):=V);
    INTEGER PROCEDURE POP;
    IF TOP>0 THEN
      (POP:=A(TOP); TOP:=TOP-1);
  INITIALIZATION:
    IF N=0 THEN ERROR;
    TOP:=0;
END;

```

An instance of this class can be generated by

S←NEW STACK(100).

The stack S would consist of an integer N, equal to 100; an array A of length 100; and an integer TOP which is initialized to zero. These attributes of S are not visible from outside, as the only way to manipulate S is by means of its operators PUSH and POP. For example, S.PUSH (7) pushes the number 7 on top of the stack, modifying the attribute TOP accordingly.

Because the internal attributes of protected objects are inaccessible from the outside, it is possible to design a class of objects which have some inherent properties that are invariant to the way a particular object is actually used. For example, the reader can easily satisfy himself that our stack does "behave like a stack". It is clear that the ability to generate information objects which have a well defined structure and behavior can contribute immensely to the reliability of programming, to our ability to prove correctness of programs, and to well structured programming in general. The reader is referred to [5] for further discussion of this important concept. Here, we will use the concept of protected objects in two ways: First, we will show that it is not compatible with conventional files. Later on we will suggest, as a solution to this problem, that the file itself can be viewed as a kind of protected object.

There are essentially two ways to store and retrieve a composite structure on a file:

- 1) We can take the structure apart, storing (retrieving) each of its components separately.
- 2) We can treat the structure as an indivisible entity, expecting the language processor to do the actual work for us.

Both these methods are not suitable for protected objects. The first method is not available because the attributes of the object are not accessible from the outside, and therefore cannot be output one by one. As to the second method, suppose that there is a stack of size N stored in the "current" position of a file F. Following the standard procedure (of, say, ALGOL-68), and supposing that N is known, we can retrieve the stack as follows: First, we generate a suitable empty structure by writing

S←NEW STACK(N).

Next, we can write something like

READ FROM F INTO S,

expecting S to be filled with information from the file. The problem here is: What if in fact the content of the file is not a stack of size N, or not a stack at all? S would then be filled with arbitrary information, which may not satisfy the "inherent properties" of a stack. For example, the attribute TOP of S may well be negative, or greater than N. Obviously, our expectations as to the behavior of S would not be satisfied.

Thus, the only available method to store and retrieve protected objects from a conventional file destroys the very essence of the concept of protected objects.

One can, of course, legislate that protected objects should not be stored on a file at all, but this would be a most unreasonable restriction. For example, such a restriction introduces an unjustified semantic distinction between an internal linear-linked-list of objects, and a sequential "intraprocess file". Such a file behaves essentially as a list. The only real difference between these two structures is one of speed, similar to the difference between a page on a disk and one in core memory, under a virtual memory system.

The inevitable conclusion from all this is that the conventional semantic-less file is inconsistent with the underlining philosophy of many programming languages.

step towards a linguistically consistent concept of file has been taken in the language PASCAL [6]: If RT is some PASCAL type, then the declaration

TYPE FT = FILE OF RT

defines FT to be a file-type. An instance of FT can be generated by

VAR F:FT.

F is a file in which only records of type RT can be stored. Thus, the domain of a PASCAL file is restricted, just as the domain of internal variables, by type specification.

The PASCAL file concept, just described, leaves a serious problem unsolved: Suppose that a file F of type FT was generated by a process P and is to be read by another process P'. How can P' handle the records of type RT, retrieved from F, if the type RT is defined in P? The structure of these records would be illegal, indeed meaningless in P' because the definition of their type is not available to it.

This problem, and its solution, are best viewed in the context of block structured languages. It is a basic principle of such languages that if an object q is an instance of type T, the "life time" of q cannot exceed that of the type T itself. The life time of a type, in turn, is identical to that of the block in which it is defined. Our problem, now, is the following: The file F which is to be used by two processes may outlive both of them. Therefore, the file-type FT, as well as the record type RT, must be defined in a block which is outer to both processes. The problem is that under most languages such a block does not exist, because a program, and any process induced by it, is commonly considered to be linguistically autonomous. This problem can be rectified by the following natural extension of the block structure concept:

Every program (in a given language) is to be viewed as if it operates within a unique outer block, of indefinite life time, to be called "the external environment", or E. E would contain, among other things the definition of various types of records, such as RT above, and of various file-types, such as FT. Obviously, there should be a way to introduce such definitions into E. It is also necessary to establish some protocol by which a program gains access to desired parts of E. (The ALGOL-like scope rules under which the entire block is visible from all its inner blocks is clearly unsuitable here). Such protocol, and the techniques which might be used for the maintenance of E, are beyond the scope of this paper, but an example will be given later.

Thus, for a file to outlive the process which generates it, it is necessary that the type definition of this file resides in the external environment E. Some standard file-types should reside in E permanently, such as a file whose records are character strings. But if one wants his file to contain records of a user defined type, and if this file is to outlive the process which generates it; then the record-type, along with an appropriate file-type, should be first defined into E.

A DATA PROCESSING VIEW OF FILES

The PASCAL-like file-type, with the elaboration discussed above, is quite satisfactory from the viewpoint of programming languages, but it does not answer the needs of data processing. When designing a file which is to carry valuable and sensitive information, it is not enough to specify the structure of the records to be stored in it. One would like to determine some ground rules under which it is to operate. A linguistic tool which is suitable for this purpose is, again, the SIMULA "class", (or rather, the protectable class as in the PDP/10 version of SIMULA). In the following section we will describe the main features of an implementation of our file concept, where a file-type, or rather, a file-class is defined by means of a SIMULA class. The file itself would be a protected object which is an instance of such a class.

The Content of a File

As was pointed out in the introduction, one of the main features of a file, which distinguishes it from a database, is the simplicity of the data stored in it. It should be simple enough not to require a complex system to handle it. To see what this means, observe that the complexity of databases is due primarily to the fact that they have to handle data which is both large and structurally complex. There is no great difficulty in dealing with large numbers of non-related records: the conventional file does that. Nor is there any real difficulty in dealing with structurally complex data, if it can be kept in main memory. It is the combination of these two properties that is difficult to handle, and which we must avoid when dealing with files.

We propose, accordingly, that the content of a file^{would} consist of two parts, to be called the "record space" (R), and the "global space" (G), defined as follows:

- (1) The Record Space (R) is an unbounded (usually large) number of records without any interrecord relations. As in the case of conventional files, only one record of R will be handled by the file at any moment in time; it will be called the "current record" of the file, to be denoted by C.
- (2) The Global Space (G) is a relatively small data structure, of an arbitrary structural complexity. It should be small enough to be wholly contained in the main memory, if necessary.

The record space is of course the counterpart of the entire content of the conventional file. The new element here is the global space. As we will see later, all of G is to be accessible during any interaction with the file. This should not contribute significantly to the complexity of the file system, due to the assumed small size of G. The global space contains information which pertains to the file as a whole, as well as information which is common to all records in R. As we will see later, G can be viewed as a kind of environment in which all the records in R are defined. The role of G in our model is best explained in the context of the dynamic behavior of a file, which is discussed next.

The Dynamic Behavior of Files

The dynamic behavior of a file is determined by the procedures built into the file-class of which it is an instance. During interaction with a file, these procedures have access to the entire global space (G) and to just one record of R. This is the "current record", C, which serves as a window that moves across R. Since G is constantly available, it can be examined and updated both by explicit user's instructions, and, behind the scenes, by various file procedures, (see figure 1).

The file procedures can be classified into the following categories:

1. The Operators of the file. This category includes the conventional file operators such as OPEN, FIND, STORE, DELETE, etc., as well as less familiar ones to be mentioned later.
2. The Internal Procedures which are not visible from the outside and which perform various activities

find the scenes, as described below.

The role of the internal procedures of a file and of its global space can be illustrated by the following classification of the activities which might be performed by the file without the explicit knowledge of its users.

a) Protection of the Integrity of a File: Various validity checking routines can be coded as internal procedures of the file-class, and invoked by the operators whenever appropriate. These procedures would usually depend on parameters, such as the permissible bounds of various components of a record. Since such parameters are relevant to all the records in the file they should be stored in G. The role of G here is crucial. In a system like ASAP [1], for example, which does not have any global space, the information which is necessary for validity checking is stored in the local space of the validity-checking-routines. This is unsatisfactory on two accounts. First, the same information item might have to be repeated in several routines. The second, more serious, problem is that such information is not subject to convenient update/retrieval procedures. In our case, on the other hand, the information in G can be manipulated by qualified users just like the rest of the file.

b) Monitoring: The importance of monitoring the activity of a file is aptly explained by Conway, Maxwell, and Morgan in (7). Here are two examples borrowed from that paper:

- (1) In an accounts receivable system someone could be responsible for monitoring the activity of a certain subset of accounts (perhaps because they are especially good customers, or because they are especially bad credit risks) and want to be informed of any transaction that affects the balance of any of these accounts.
- (2) In an inventory system a particular item might be depleted and back-ordered and a user might wish to be notified as soon as replenishment takes place.

Of course, such monitoring must be done without the explicit knowledge of the users who interact with the file. In our system, the monitoring will be performed by special objects called demons which are activated by the file system under a specified set of circumstances. The procedural part of these demons must be pre-defined (as part of the definition of the file), but demons can be activated and deactivated by qualified users during the lifetime of the file.

c) Maintenance of the File Status and History: The global space can be used to carry information about the current status of a file and about its history. Such information can be computed routinely by the various operators and by some of the demons mentioned above. Examples of information that one may want to maintain in G are: the total number of records in the file; various aggregates such as the current balance in an accounting file; some statistics about the past activities of the file, etc. The only limit here is the size of G which must stay fairly small.

d) Access Control: G can be used to store information about the access rights of various classes of users, grouped into structures called "user profiles". When a file is opened, an appropriate user profile is selected to be later consulted by the various operators. In our current design there is a standard access control mechanism based on security levels and compartments (as in ADAPT-50 [8]). In addition, there are built-in hooks on which one can hang procedural access control of arbitrary generality, as in [9]. Note that the manipulation of G as well as that of R should and can be under access control, this is true, in particular, for the manipulation of the user-profiles themselves.

e) Transformation of Data: Several types of data transformation may be necessary. Here are the most important of them:

1. Enciphering and deciphering of stored data for security purposes. The keys which are necessary for this transformation can be stored in G and would be subject to user interaction under the restriction of the access control mechanism.
2. A transformation between the logical and physical representations of the records in R. Such a transformation may, in particular, save space by performing data compaction and by eliminating redundancy.
3. Transformations between an actual record in the file and users' views of it (these are the schema-subschema transformations, as in the DBTG proposal).

To illustrate the use of our files, suppose that there is a file class PAYROLL in the environment E. Also in E, there is a class EMPLOYEE, which defines the structure of the records in payroll-classes; and the class SUMMARY which defines one of the structures in the global space G of payroll-files. (G may have other parts which will not concern us here.) Suppose also that there are in the system two "payroll files" called "part-time" and "full-time". (The term "payroll files" means that these files are instances of the class PAYROLL).

Now, a user who wishes to interact with any payroll file must have in his program the declaration

EXTERNAL PAYROLL, EMPLOYEE, SUMMARY...

which incorporates these classes into the user's program.

An internal data structure which can be used for the interaction with a payroll file can now be generated by

F ← NEW PAYROLL

F will be called a "file anchor". To associate F with a specific payroll file, the file "full-time" say, one writes:

F.OPEN("FULL-TIME",USER-PROFILE,PASSWORD).

This operation opens the named file, if an appropriate password is provided for the requested user-profile.

The file "full-time" can now be manipulated and interrogated by applying various file operators to the anchor F. For example, assuming that F is an indexed file, and that the index is the name of an employee, one can retrieve a specific employee record by

e ← F.FIND("name")

The record e would be an instance of class EMPLOYEE and can be used as such. (Recall that the definition of class EMPLOYEE is included in the user's program).

An employee record e which is generated by the user can be stored in the file by

F.STORE(e).

Let us use the STORE operation to illustrate the behind-the-scenes activity of the file. The following are the actions which might be invoked by the STORE operator:

- (a) The validity of e is checked, as well as the right that the current user has to store such a record.
- (b) The various "demons" which are supposed to monitor the store operation are invoked.
- (c) The record e may be transformed into an internal representation, which is, presumable, more economical.
- (d) The record is enciphered.
- (e) Finally, the transformed and enciphered record is stored, using the key defined in it, if any.

Of course, the nature of these activities, and whether they exist for this particular file, depends on the class PAYROLL, (see also next section).

The global part of the file can also be retrieved and updated. Here is an example. We already assumed that our G includes a record which is an instance of class SUMMARY. This record may have an attribute such as #employees, which is to be maintained by the various "demons" of the file. The summary record can be retrieved by

s ← F.GET-GLOBAL("summary"),

where the returned object 's' is a summary record. The number of employees in the file are now accessible by s.#employees.

Until now we discussed only the standard file operators, which exist for any file. One can also define operators which are specific to a certain file-class. For example, it may be very convenient to define into our employee file an operator REPORT which prints a report about an employee, or an operator PRINT-CHECK(k) which prints a paycheck for \$k, for the current employee. The incorporation of such data dependent operators into the file itself may save a lot of coordination efforts between programmers and programs. It is a key to "data independent" file processing.

As a final example let us show how a new payroll file can be generated. The first step is, again, a generation of an anchor

F ← NEW PAYROLL

Now we have to load F with the desired initial G, a step which will not be described here due to lack of space. Next we write

F.GENFILE(filename, access-mode,...).

This operator generates a new payroll file whose name is the first parameter and whose access-mode is identified by the second parameter. It is assumed here that we have a repertoire of access modes available to us, such as "sequential", "ISAM", etc. The specific access mode of a file must be determined when the file is generated. Note, however, that to a large extent, the interaction with an existing file may, and should, be independent of its access mode.

On the Definition of a File-Class

As was already pointed out, a file class is defined by means of a module which determines both the structure of the file's content and its behavior. The problem here is that some of these characteristics are peculiar to a given file, while others are common to all files and should be defined once and for all, for a given language. Thus, a file-class should be defined jointly by the language and by an individual user. Fortunately, there is in SIMULA a ready made tool for such joint definitions. It is the ability to extend a class as follows: A given class C1 can be extended by the module

C1 CLASS C2 BEGIN...END

The resulting class, C2, consists of all the attributes of C1 together with all those defined inside the BEGIN...END bracket. C2 is thus an "extension"⁽¹⁾ of C1, while C1 is called the "prefix" of C2. We will also refer to C2 as a "C1-class".

Two properties of this linguistic device of SIMULA are important for us here. First, a class can have any number of different extensions. Secondly, certain parts of a prefix can be redefined in its extensions.

The class-extension capability of SIMULA suggests the following approach to the definition of files: First, there should be, in the environment, a standard class called FILE which contains all the structures and procedures that are common to all files. For example, operators like FIND, STORE, OPEN... would be defined in FILE. Also in FILE, there would be various default procedures which could be redefined later. A specific file-class would be defined as an extension of the class FILE, to be called a "semantic extension" of FILE. This extension would contain everything which depends on the identity of a specific class of files. The only mandatory part in the semantic extension is the specification of the type (class) of the records in R. Indeed, a conventional type of text-file can be defined simply by the following module:

FILE CLASS TEXTFILE;

BEGIN TEXT RECORD END;

Note that the G part of this file is empty. Optionally, however, one can put many more details into the semantic extension of FILE. For example, the specification of G, various validation routines, access control procedures, enciphering procedures, etc. All that can be done in a very modular way since the general framework and the control structure are already defined in FILE. Unfortunately, space limitations do not allow us to present a complete example of a definition and use of a file.

CONCLUSION

The current state of the art of information processing tends to link capabilities such as protection of data integrity and its confidentiality, monitoring the interactions of users with the data, etc., with large database systems. This is most unfortunate since such capabilities are frequently vital for modest applications which do not have the degree of data complexity to justify the use of large database systems, and which may not be able to afford them. In this paper we tried to demonstrate that what we called "intelligent archivist's capability" can be implemented as a standard feature of a programming language, without the large overhead associated with database systems.

The paper is based on an actual implementation under the SIMULA language. It is still an open question, however, as to how to implement an equivalent file system into a more common language, like COBOL.

ACKNOWLEDGEMENT

I wish to thank Professor Irving Rabinowitz and Professor David Levine for reviewing this paper and for their very helpful suggestions, and to Mss. Marti and Trudi for typing this manuscript, and for correcting a lot of spelling mistakes.

(1) We are taking some liberties with the SIMULA concepts and terminology. For example, in SIMULA, C2 is called a subclass of C1.

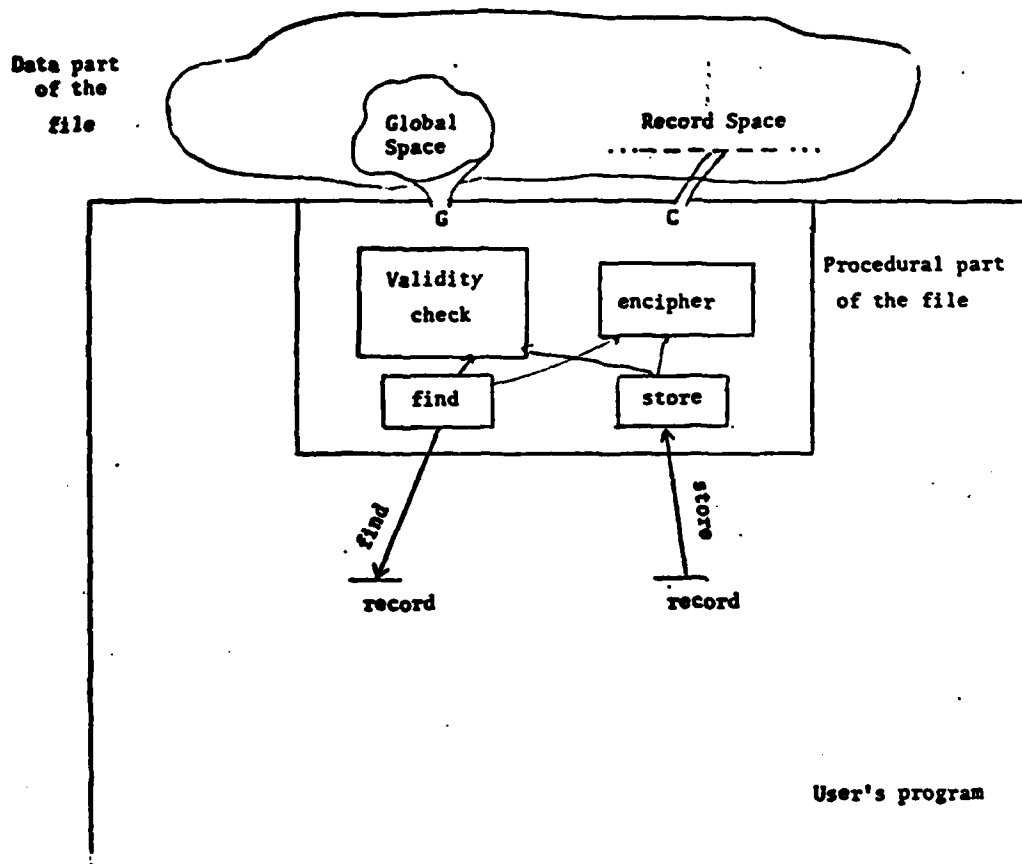


Figure 1: An illustration of the file structure .

A user communicates with a file by means of its operators. These, and the other file procedures have an access to the "current record", which moves across R like a window, as well as to the entire "global space" of the file.

REFERENCES

1. R.W. Conway, W.L. Maxwell, H.L. Morgan; ASAP 2.0 system ref. man. Compuvisor Inc., Ithaca N.Y. (1971).
2. P. Naur, "Revised Report on the Algorithmic Language ALGOL 60", CACM 6,1 (1963).
3. "Informal Introduction to ALGOL 68", Lindsay and Vander Meulen, North Holland, (1972).
4. J. Palme, "Protected Program Modules in SIMULA 67", Tech. Rep., Res. Inst. of Nat. Def. Stockholm, (July, 1973).
5. B. Liskov, S. Zilles, "Programming with Abstract Data Types", proceeding of the ACM SIGPLAN Conf. on very high level languages, (April 1974).
6. N. Wirth, "The Programming Language PASCAL", Acta Informatica 1 (1971).
7. R.W. Conway, W.L. Maxwell, H.L. Morgan, "A Technique for File Surveillance", IFIP Congress (1974).
8. C. Weissman, "Security Control in the ADAPT-50 Time Sharing System", proceeding of the FJCC, (1971).
9. L.J. Hoffman, "The Formulary Model for Flexible Privacy and Access Controls", Proc. of the FJCC (1971).
10. O.J. Dahl, B. Myhrhaug, K. Nygaard, "The SIMULA 67 Common Base Language", pub. s-22, Norwegian Computing Center, Oslo (1970).

SOSAP-TR-24

May 1976

A SEMI-LATTICE MODEL FOR SECURE INFORMATION FLOW

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

A Semi-Lattice Model for
Secure Information Flow.

by

N. Minsky

Abstract

The Lattice model for secure information flow proposed by Dorothy Denning is found to be unjustified in a number of cases. A modification to this model is proposed, which allows for controlled violations of the lattice discipline.

Key words and phrases: protection, security, information flow, security classes, enciphering, lattice

CR categories: 4.35

In a recent paper, [1] Dorothy Denning proposed a very attractive lattice model for "secure information" flow. Although this model successfully handles many aspects of the problem at hand, it fails to represent certain important flow patterns of information, which do not behave like a lattice. These non-lattice properties will be identified, and a way to modify the original lattice model to cover them will be proposed. We start with a brief summary of the lattice model, assuming that the reader is familiar with Denning's paper.

The information flow model, FM, is defined by

$FM = \langle N, P, SC, \rightarrow, \oplus \rangle$, where:

P is a set of processes (which will not concern us directly here),

$N = \{a, b, \dots\}$ is a set of objects, and SC a set of security classes.

Every object $a \in N$ is bound to a security class $\underline{a} \in SC$, although the binding is not necessarily permanent.

" \rightarrow " is a relation defined on SC , called the flow relation. It has the following meaning: For $A, B \in SC$, the relationship $A \rightarrow B$ means that information in class A is allowed to flow into class B . The algebraic structure $\langle SC, \rightarrow \rangle$ is assumed to be a partially ordered set.

" \oplus " is a binary operator defined on SC . The following is assumed about it: Let a, b be objects in N , and let $\underline{a}, \underline{b}$ be the security classes associated with them. It is assumed that the security class associated with $f(a, b)$, for any function f , is $\underline{a} \oplus \underline{b}$. (We will find below that this assumption is not always justified).

Now, it is claimed in [1] that under a set of assumptions which follow from the semantics of secure information flow, the algebraic structure $\langle SC, \rightarrow, \oplus \rangle$ forms a lattice in which " \oplus " is the "least-upper-bound" operator. One of the assumptions which supports this claim is the following:

For any $A, B \in SC$ the following relations are satisfied (in the "real world" of secure-information-flow, that is).

$$A \rightarrow A \oplus B \text{ and } B \rightarrow A \oplus B$$

"Without this property", it is said, "we would have the semantic absurdity that operands could not flow into the class of the result generated from them". This, however, is not absurd at all, as the following example illustrates.

Consider the function encipher (plaintext, code). The two parameters of such a function would usually have high security classification while the security level of the outcome of the encipher function is relatively low. Indeed the whole purpose of enciphering is to generate an image of a confidential plaintext which can be sent via public channels, or be stored on loosely protected files. The outcome of encipher must therefore have a lower security classification than its arguments. Thus, we do not have in this case the relation

$$\text{plaintext} \rightarrow \text{plaintext} \oplus \text{code}$$

Moreover, this example shows that one cannot expect the security level of $f(a,b)$ to be the same for all functions of f .

The crux of the matter is that the lattice discipline does not allow any reduction of the security classification, where such a reduction is frequently vital and justified. For instance, our enciphering example is based on the beliefs that it is impossible, or very difficult, to figure out the plaintext from its coded version, there is thus no reason for the latter not to have a lower security class than the former. As another example of reduction, consider a function summarize(f) which produces a summary of the content of a file f. Even if every record of f may be of high security classification, the function summarize may be carefully programmed to produce only an insensitive summary of it. There is no reason not to give this summary a low classification. As a final example, there should be somebody in any organization who has the power to declassify items of information. Such power must be reflected in the computerized information system, but it is not compatible with the lattice discipline.

In spite of all this, the lattice remains an eminently suitable model in many ways, and should be retained whenever possible. I am proposing, therefore, a "semi-lattice" model for secure information flow, which is based on Denning's lattice, but accommodates some controlled violations of it. The permitted lattice violations are defined by rules of the form

$$c_0 \leftarrow \langle f, c_1, \dots, c_m \rangle,$$

where f is an identifier of a given n-ary function, such as our encipher function above. c_i , for $i=1, \dots, n$ are the security classes of the

parameters of f , and c_0 is the security class of the outcome of f . There is no a priori relation between c_0 and c_1, \dots, c_m . c_0 is determined arbitrarily by the rule above. Consider, for example, the case illustrated in Figure 1. Suppose that A_2 is the class associated with the enciphering code and A_1 is associated with the plaintext to be enciphered. In addition to the flow permitted by the lattice, represented by solid lines in Figure 1, suppose that there is the rule:

$$A_0 \leftarrow \langle \text{encipher}, A_1, A_2 \rangle.$$

This rule means that the outcome of encipher, when applied to parameters of classes A_1 and A_2 , should be associated with the lowest class, A_0 . (In Figure 1 such rules are represented by diamond shaped figures, and the corresponding lattice-violating flows by dashed links). Another violation, in this example, is permitted by the rule

$$A_0 \leftarrow \langle \text{declassify}, A_3 \rangle.$$

Clearly, the use of some of these lattice-violating functions, such as declassify, must itself be restricted, lest it be abused. Therefore, a model such as this must reside in a tightly controlled computing environment. For example, the formalism proposed in [2] is based on rules which are very similar to the ones used here and might provide a suitable environment for the semi-lattice model.

Finally, it should be pointed out that the various enforcement techniques proposed in [1] are still valid for the semi-lattice model described here.

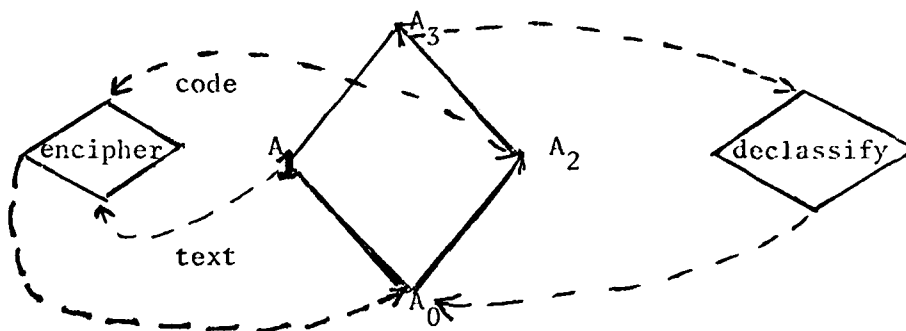


Figure 1: An example of a semi-lattice

The solid links between the security classes A_1, \dots, A_4 form a lattice. The diamond shaped boxes represent lattice-violating functions, and the dashed links represent lattice-violating flow of information.

References

1. Denning, F.D. "A Lattice Model for Secure Information Flow"
Comm. ACM 19, 5(May, 1976), 236-243.
2. Minsky, N. "Intentional Resolution of Privacy Protection in
Database Systems" Comm. ACM 19,3 (March, 1967) 148-159.

SOSAP-TR-27

September 1976

PROTECTION IN PROGRAMMING LANGUAGES BY OPERATION-CONTROL

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

PROTECTION IN PROGRAMMING LANGUAGES BY
OPERATION-CONTROL

Abstract

Protection in programming languages has to do with dispersion of privileges among the various modules of the program. This is being done in most languages by means of scope rules. Recently it has been proposed to incorporate into languages, an access-control facility which is modeled after the capability-based-protection mechanism provided by some operating systems. This paper proposes a further enhancement of our ability to disperse privileges, by the introduction of a new control-object called activator, which represents privileges with respect to an operator. The resulting protection scheme, to be called operation-control, is based on the access-control facility, but it is more general and often more efficient than the latter.

1: INTRODUCTION

By the term "protection" we mean, the ability to impose restrictions on a program, restrictions which cannot be violated by the program itself regardless of the actual code in it. These restrictions may be imposed on a program-module from the outside; or, they may be voluntary restrictions introduced by declarations, say. Such restrictions are the basis for effective modularization, and they facilitate program-verification, either manual or automatic. Protection is therefore essential for the production of reliable software. It is particularly crucial for the design of large systems with many modules, where the controlled sharing of information is imperative.

One may distinguish between two types of restrictions, to be called "semantics-based-restrictions" and "privilege-based-restrictions". The objectives of the former type of restrictions is to guarantee the semantic integrity of the various data-structures manipulated by a program, and to prevent meaningless operations from being carried out. The privilege-based-restrictions, on the other hand, have to do with a dispersion of privileges among the various modules of a program, in order to limit the set of legal operations which can be carried out by any given module. Although the boundary between these two types of restrictions is not sharply defined, the distinction between them would give us a convenient frame of reference. This paper, in particular, is concerned primarily with the second type of restrictions. But before we get to that, a brief review of the current state of the art of protection in languages is in order.

Most of the semantics-based-restrictions facilitated by programming languages are associated with the concept of type. A notable example is provided by languages like CLU [Lis74], ALPHARD [Wul75] and some versions of SIMULA [Dah72, Pal73]. In these, and other languages, the type definition, of a given type t, in addition of being a template for all instances of t, (which we will call t-objects), also contains a set of procedures to be called t-operators (*). These operators have the exclusive direct access to t-objects. That is to say, the only way to observe or update a t-object, from any place in a program outside of the module which defines t, is indirectly, by means of the t-operators. The great usefulness of this concept for reliability of programming was aptly described by Liskov and Zilles [Lis74].

Turning now to the privilege-based-restrictions, the best known examples of these, are the "scope-rules" which form the block-structure in many languages. The effect of these, and other more general scope-rules [Wul74, Dij76], is to restrict the set of objects which are accessible to a given module. Note, however, that the accessibility of an object, as determined by the scope rules, is usually an all-or-nothing matter. Namely, if an object b of type t is accessible to module M, then every t-operator can be applied by M to b. In order to facilitate controlled sharing of information, it has been recently proposed, by Jones, Liskov, Wulf and others [Jon76, Wul76], to introduce into programming languages an access-control facility which is based on the capability-based-protection technique, originally developed for operating-systems. Here are the essentials of this

(*)We will use the terms "procedure" and "operator" interchangeably.

technique:

A sharable-object (usually allocated on the heap) can be accessed only by means of a special type of object which we will call a ticket, (the more common name, "capability", would not be appropriate in the context of this paper). A ticket for an object of type t is essentially a pair $(b:t;r)$ where b is the identifier of the object, and r is a subset of a finite set of symbols

$$\text{rights}(t) = \{r_1, \dots, r_n\}$$

associated with the type t. These symbols are called the "rights" with respect to t-objects. Now, the point is that the set of operators which can be applied to an object, depends on the rights contained in the ticket which is used to address the object. Thus, a ticket (b:t;r) contained in module M, represents the privileges that M has to object b. (For more details about this protection scheme, and about its incorporation in a programming language, the reader is referred to [Jon76].)

In this paper we will argue that tickets are not always sufficient for the representation of privileges. In particular, they are not very suitable for the representation of value-dependent privileges, or for the control of interaction between objects. In order to make up for these and other limitations of the tickets we will introduce an object called, activator to serve as a reference to an operator, and to represent privileges with respect to this operator. By the phrase "privileges with respect to an operator" we mean, loosely speaking, the conditions under which the operator, addressed by the given activator, is allowed to be invoked. The essentials of a protection scheme under

which both activators and ticket are used for the representation of privileges is introduced in the next section. Being based primarily on activators, which are anchored on operators, the proposed protection scheme will be called "operation-control" (or OC); as opposed to the term "access-control" (AC), for the protection facility based on tickets, which are anchored on objects. The two facilities are compared in section 3. The rationale for the introduction of activators is also discussed there.

2: THE PROPOSED PROTECTION FACILITY

2.1: The Activator

Definition: Let o be an operator of degree k (with k formal-parameter). An activator for o , or an " o -activator" is the following construct:

$$\langle o, n_1:p_1, \dots, n_k:p_k \rangle \longrightarrow p_o$$

Here o is an identifier of the operator; p_i , for $i=1, \dots, k$ is a condition on the i -th operand of o , to be called "operand-pattern"; and p_o is a condition on the outcome (result) of o to be called "outcome-pattern" (*) (if o does not have an outcome, then the part " $\longrightarrow p_o$ " does not appear in the activator). The symbols n_1, \dots, n_k are names of the respective operands; these names are optional. (A more general concept of activator is introduced in [Min76B, section 3.6]).

(*) Whenever we would not wish to distinguish between these two types of patterns, we will use the term "activation-pattern", or just "pattern", for either of them.

Now, it is assumed that in our language, an operator is invoked by the phrase

$$A(q_1, \dots, q_k),$$

where A is an o -activator for some operator o , and q_i are the actual-parameters for o . This phrase is to be evaluated as follows: First, the conditions p_i of the activator A are evaluated on q_i for $i=1, \dots, k$. (Namely, the condition p_i has one free variable, which is to be bound to q_i). If all these conditions are satisfied, in which case we will say that "the operands match the respective patterns of A ", then the operator o is applied to q_1, \dots, q_k . Upon return, the outcome (value) of o , if any, is checked. If it does not match the outcome-pattern p_o of A , it will not be communicated to the program, and an error condition may be raised. (This checking, or pattern-matching, may be performed either at run-time or at compile-time, depending on the structures of the patterns as well as on other properties of the language).

Thus, the availability of a specific o -activator to a program-module determines the conditions under which the operator o can be invoked by this module.

Note the similarity between an o -activator and the formal-parameters-specification (or FPS) of the procedure o : Both determine the legal set of operands of o . There is, however, an important difference between these two. While there is just one FPS per operator, we will see below that there may be several different o -activators for the same operator o . In this sense, our activators are similar to the "procedure-closure" in EL1 [Weg74]. However, unlike

the Ell procedure-closures, our activators differ from each other not in the degree of binding of free variables, but in their activation patterns.

Before committing ourselves to any specific structure of the activation patterns, we will first discuss some general properties of the activators.

Let p be an activation pattern. We define range(p) to be the set of all possible objects which can be matched to p , namely, which satisfy the condition p . Let p and p' be two activation-patterns. We will say that p' is weaker than p or, equivalently, p is stronger than p' , if

$$\text{range}(p') \subseteq \text{range}(p)$$

We will say that p' is strictly weaker than p if

$$\text{range}(p') \subset \text{range}(p)$$

A pattern p' which is weaker than p will also be referred to as a reduction of p .

Let A be an activator of order k . We define range(A) to be the set of all possible $(k+1)$ -tuples (q_1, \dots, q_k, q_0) of objects, such that q_i matches the activation patterns (or satisfies the condition) p_i , for $i=1, \dots, k$, and for $i=0$.

Let A and A' be two o -activators for a given operator o . We will say that A' is weaker than A (or, equivalently, A is stronger than A') iff

$$\text{range}(A') \subseteq \text{range}(A).$$

We will say that A' is strictly weaker than A iff

$$\text{range}(A') \subset \text{range}(A),$$

such an A' will also be called a reduction of A . It is clear that this last relation is satisfied between A and A' iff all the activation-patterns of A' are weaker than the corresponding patterns of A , and at least one of the patterns of A' is strictly weaker than that of A .

As to the generation of activators, the following is assumed: A new activator can be generated in the following two ways:

- a) When a new operator o is created, an o -activator is generated with it. It will be called the primary o -activator.
- b) Given an activator A , one can generate from it an activator A' which is weaker than, or equal to, A . A' will be called a derivative of A .

The following properties of the activators follow immediately from the above:

- a) The set of all o -activators, for a given operator o , is partially ordered with respect to the relation "stronger".
- b) Every activator is stronger than all its derivatives
- c) The primary o -activator is the strongest o -activator.

Note that the generation of new activators, and their distribution among the various modules of a program, may be performed either at

compile-time or at run-time depending upon the general architecture of the host language. In the former case, activators are declared. They serve primarily as a self imposed restriction, on the part of the programmer, as to the use which he intends to make of the operator in a specific module. In the latter case, one would have "activator-values" which can be stored dynamically in "activator-variables", much in the same way as some languages have procedure-values and procedure-variables. One would also need special operators which can generate a derivative of a given activator, and transport activators from one place to another, at run-time.

2.2. The Activation Patterns

As defined above, an activation pattern p_i is a condition which must be satisfied by an object, which is either the i -th operand, when $i=1, \dots, k$; or the outcome of the operation, when $i=0$. For the most parts there will be no need to distinguish between these two cases, and we will be talking simply about an "activation-pattern" p and the object which matches it. Whenever necessary, however, we will distinguish explicitly between an "operand-pattern" and an "outcome-pattern". We will now describe a specific structure of activation patterns.

An activation pattern p , to be denoted by

$$[I; R; V]$$

is a conjunction of the following three predicates, to be called the components (sub-patterns) of p :

I - the identity-based pattern, is a condition on the type and identity of an object to be matched with it.

R - the privilege-based pattern, is a condition on the privileges, or "rights", represented by the operand-ticket. (This sub-pattern is applicable only if the operand is a sharable object).

V - the value-based pattern, is a condition on the value, or the state, of the operand itself.

The only mandatory part of p is I. Each of the other components may be empty, in which case it will be assumed to be identically TRUE. The reduction of the pattern p, namely, the reduction of range(p), is performed by a reduction of at least one of its subpatterns.

We will now discuss the structure of each of these three sub-patterns, and the ways by which they can be reduced.

2.2.1 Identity-based patterns: I, which is the only mandatory component of a pattern, has one of the following two forms.

t - which matches any object of type t.

b - which matches only a specific object b of type t, b being an object-identifier. This form is applicable only when t is a sharable type and only in the case of operand-pattern.

A subpattern t can be reduced in two ways:

a) It can be changed into b, where b is an identifier of a specific object of type t.

- b) If there is a type t' such that t includes t' , then t can be changed to t' , ("type-inclusion" exists in a number of languages, for example in Simula [Dah72]. "Inclusion" in meant here in its set theoretical sense).

The sub-pattern b cannot be reduced any further.

The type t , identified by I , determines the structure of the other two components of the pattern which will sometimes be denoted by $V(t)$ and $R(t)$.

2.2.2. Value-based patterns: V , which is the third component of a pattern, is a condition on the value of the operand. Although we consider value-dependency to be a vital aspect of protection, one cannot always expect to be able to base the decision as to the legality of an operation on the value of an arbitrary attribute of an object. This is due to the following reasons:

- a) It happens that an attribute of an object is not directly observable. Such are the items stored inside a stack, for example.
- b) It may happen, in certain types of objects, that the mere act of observation of an attribute of an object introduces a change in the object itself. As an example of this "quantum mechanical effect", the only way to observe the top of a stack may be to pop it up. Obviously, we cannot base our pattern matching on such a "volatile attribute".

For these, and other reasons which include efficiency, we now introduce

the notion of "facade";

The component V (or rather $V(t)$) of a pattern, can depend only on a set of attributes, called the facade(t), which are explicitly declared (as part of the type-definition) to be useable for this purpose. In particular, for a given type t , if $\text{facade}(t)$ is empty then $V(t)$ must also be empty. By convention, the facade of a primitive scalar type, such as real and integer, will be its value. We are ready now for the definition of the functional form of V .

$V(t)$ is a conjunction $v_1 \& v_2 \& \dots \& v_n$, where each v_i is an arbitrary predicate over the set of attributes in $\text{facade}(t)$ of the operand.

The reason for the conjunctive form of V is that it allows the following simple reduction technique: V is reduced by appending a conjunct to it.

One may want to impose various additional restrictions on the functional form of V , for efficiency reasons. Such restrictions will not concern us here.

Example 1: Let doc be a type of sharable objects which carries documents in a military information system. Suppose that a document is represented by a triple: (data:text, security:integer, cat:text), where the phrase data:text, for example, refers to an attribute called "data" of type "text". Here, "data" stands for the body of the document, while "security" and "cat" are the security-class and the category associated with the document. The latter two are the traditional security parameters in military establishments, [Wei 69]. In order to use these attributes for protection in our system, we

define:

$$\underline{\text{facade}(\text{doc})} = \{\text{security:integer, cat:text}\}$$

Now, let read be a doc-operator which, when applied to a document displays its contents. Suppose that the primary read-activator is

$$\text{READ} = \langle \text{read}, [\text{doc}] \rangle.$$

which can be applied to any documents The following derivatives of READ are less powerfull: The activator

$$\text{READ1} = \langle \text{read}, [\text{doc}; ;\text{cat}=\text{"navy"}] \rangle$$

can be used to read any Navy-document. (Note that the extra semicolon in READ1 indicates a missing R part in the pattern). The following reduced derivative of READ1:

$$\text{READ11} = \langle \text{read}, [\text{doc}; ;\text{cat}=\text{"Navy"} \ \& \ \text{security} \leq 3] \rangle$$

can be used to read only Navy documents whose security level is not higher than 3. Finally, the activator

$$\text{READ12} = \langle \text{read}, [\text{d}; ;\text{cat}=\text{"navy"}] \rangle$$

can be used to read a specific document d, provided that it is a Navy-document. Of course, a module may have access to several read-activators, which would allow it to read several such subsets of documents. Note however, that the availability of an activator is, in itself, not sufficient to read any specific document. For this one has to have tickets for the appropriate documents.

2.2.3: Privilege-based patterns:

To support the type of access-control mentioned in the introduction,

the remaining component of the activation pattern, R , is a condition on the rights which are contained in the operand-ticket. More specifically, let the type t identified by the I-component of the pattern be a sharable type, then R (or $R(t)$) has the general structure

$$R = s_1 \& s_2 \& \dots \& s_k$$

where each s_i is a right with respect to type t , i.e. all s_i are members of $\text{rights}(t)$. As to the interpretation of this sub-pattern, we will have to distinguish between the case of an operand-pattern and that of an outcome-pattern, starting with the former.

Consider the operand-pattern

$$p = [I; s_1 \& s_2 \& \dots \& s_k; V]$$

For an operand to be matched to p , it must be represented by a ticket

$$c = (b:t; r)$$

where b and t satisfy the sub-patterns I and V , and r includes all the rights s_1, \dots, s_k which appear in p . Thus R is a set of rights which are required from the matching ticket. The reduction of the sub-pattern R is performed by adding rights to it, thus imposing stronger requirements on the corresponding operand.

As an example, consider again our document case. Let

$$\text{rights}(\text{doc}) = \{\text{observe}, \text{update}\}$$

We will now assume that there are two doc-operators: read and update. The primary-activator of read is now

READ = <read, [doc;observe]>

that of update is

UPDATE = <update, [doc;update],[text]>

where the second operand, which must be a text-object, is to be inserted into the document. Consider a module M which has these two activators in its domain, together with the tickets:

(d1:doc; ALL), (d2:doc; observe), (d3:doc)

The document d1 can be both observed and updated by M; d2 can be only observed, because it cannot be matched to the first activation pattern of UPDATE; and the document d3, which has no rights in it, can be neither read nor updated by M.

Note that the above activators may be reduced as before. For example, the following derivative of READ

READ1 = <read, [doc;observe;security<2] >

allows its holder to read any document whose security level is smaller than 2, provided that he has a ticket with the "observe" right in it.

Next, let us discuss the case of outcome-patterns. Consider the activator

A = <...> → [t; s1&s2&...&sk; V]

where t is a sharable type.

The outcome-pattern of A means, first, that the outcome of an operation

security-level and category, and gets a ticket for the generated object with all its possible rights. Suppose now that a certain module M is to be allowed to generate only documents with the category "navy", and that these documents should be unchangeable. For this purpose M should have the following reduction of GEN-DOC.

GEN-DOC-1 = $\langle \text{gen-doc}, \dots, \text{cat:}[\text{text};; \text{value} = \text{"navy"}] \rangle \longrightarrow [\text{doc}; \text{observe}]$

Note that documents generated by GEN-DOC-1 can never be changed because there can be no tickets with the "update" right for them. (This observation is based on the assumption that there is no way to add rights to a ticket, as it is the case under the protection scheme proposed in [Min76b], and it is almost the case under the scheme proposed in [Jon76]).

2.3: The control-objects

We can summarize the proposed scheme as follows: the dispersion of privileges among the various modules of a program is represented by the distribution of two types of objects, the activators and the tickets. We will refer to both as control-objects. There is a certain symmetry between these two types of control-objects: Just as a ticket represents privileges with respect to an object, an activator represents privileges with respect to an operator. Although the term "privilege" has different connotations in the two cases.

Note the complementary nature of the activators and the tickets: An activator is in a sense, the means which one has to carry out an operation, while a ticket represents the opportunity to apply an operator to a specific object. One needs both in order to actually

perform an action. For example, the activator `<read, [doc;;cat="navy"] >` is the means to read Navy documents, but in order to read any specific document, one must have an access to it, via a ticket, in our case.

Since the distribution of the control-objects determines what every module of a program can do, it is vital to have a tight control over the generation and transport of these objects. The essentials of such controls are discussed in [Min76b], but they are yet to be adapted to a programming language environment.

3: DISCUSSION

There are two dimensions along which a protection facility should be evaluated: The ease of its enforcement, and its contribution to our ability to impose a desired policy. (By the term "policy" we mean a certain distribution of privileges among the various modules of a program). In order to have a specific frame of reference for our discussion we will compare the operation-control (OC) facility in which both activators and tickets are used for the representation of privileges, with the access-control (AC) facility which is based primarily on tickets.

3.1: The enforcement mechanism

The enforcement mechanism which is necessary for the OC facility is essentially identical to the checking of procedure-parameters performed by most languages. The only difference is that the

conditions built into an activator may be more complex than those allowed in the formal-parameter-specifications of a procedure. In particular, we are allowing for value-dependent conditions, which in most cases could not be evaluated at compile time. Obviously, there is a price to be paid for the evaluation of more complex conditions. However, this price is incremental. In particular, the enforcement mechanism which is needed for activators which have no value-dependent patterns is no different than the enforcement mechanism needed to support the AC facility.

3.2: Why activators?

In this section we will discuss the limitations of tickets alluded to in the introduction. We will also see how the use of activators makes up for these limitations.

3.2.1: interactions and their control: The privileges which are represented by a ticket determine which operators can be applied to the object addressed by it. This, however, is not always sufficient to characterize all possible manipulations of objects: There may be operators which involve several objects, and which cannot be decomposed into a sequence of legal operations on the individual objects. Such an operator will be called an interaction. Now, it has been shown in [Min76b] that in a system with interactins there are policies which cannot be imposed purely by means of tickets. To illustrate this difficulty consider the following example:

Let appoint(e,j) be an interaction defined in a corporate information-system, which appoints employee e to job j. Let E1 and E2 be two sets of employees, and let J1, J2 be sets of jobs. Suppose that a module M is to be allowed to appoint only employees from E1 to jobs from J1, and employees for E2 to jobs from J2. suppose also that this restriction is specific to M and does not necessarily apply to other modules. A moment of reflection would reveal the difficulty of conferring exactly this privilege on M purely by means of tickets. There is no such difficulty under the OC scheme, however. It would be sufficient to give M the two activators.

<appoint, [employee;;vel], [job;;vj1] >

<appoint, [employee;;ve2], [job;;vj2] >

provided that vel and ve2 are predicates which are satisfied only by members of the sets E1 and E2 respectively. Similarly, vj1 and vj2 should identify J1 and J2 respectively.

3.2.2: Value-dependent restrictions: The access-control protection scheme was never intended for value-dependent restrictions [Jon75]. Indeed, the only way to impose such restrictions, strictly by means of access control, is by a suitable value-dependent distribution of tickets (see [Min76b]). As we will demonstrate later by an example, such an implementation is very costly and error-prone, particularly because this distribution must be changed with the values on which the policy depends. Of course, value dependency is very natural under our OC scheme, as was demonstrated by the examples in section 2, and will be illustrated again by another example, below.

3.3: The complementarity of tickets and activators.

The rationale for using activators is not just to make up for limitations of the tickets. As was mentioned in section 2.3, the representation of authority structures requires two types of control-objects, to represent "means" and "opportunities", both of which are necessary in order to perform an action. These complementary roles are played by the activators and by the tickets.

3.4 An example:

In order to compare our operation-control with the access-control scheme, we will now discuss the implementation of a specific policy by means of both. The example to be discussed is a generalization of an example which has been used in [Jon75] to demonstrate the features of the AC scheme. Here we will show that the same situation can be handled much easier and more efficiently under the OC scheme.

Let memo be a type of objects which carry memoranda. Suppose that in addition to the text itself, which can be retrieved by the operator read, every memo-object has a set of attributes

$$X = \{x_1, \dots, x_n\}$$

associated with it, where all x_i are boolean variables. We will say that a memo m "satisfies a certain attribute x_i " if $x_i(m) = \text{TRUE}$. Suppose also that for every subject(*) S there is a set $Y(S) = \{y_1, \dots, y_k\} \subseteq X$ of memo attributes which determine the set of

(*)Following the terminology used in the operating system protection literature, we will use here the term "subject" as a synonyme for a "module".

memos which should be readable by S , according to the following policy:
 S should be allowed to read all memos, and only such, which satisfy all y_i in $Y(S)$.

An OC-implementation of this policy is the following.

Let

$$\text{facade}(\text{memo}) = \{x_1, \dots, x_n\}$$

$$\text{rights}(\text{memo}) = \text{NULL}$$

and let the primary read-activator be

$$\text{READ} = \langle \text{read}, [\text{memo}] \rangle$$

Suppose that a subject S is given only the following reduction of READ.

$$\text{READ1} = \langle \text{read}, [\text{memo}; ; y_1 \&, \dots, y_k \&] \rangle$$

Suppose also that the set of tickets $\{(m:\text{memo})\}$, one for each memo-object in the system, is stored on a file dir from which all subjects can copy tickets. It is obvious that the desired policy is satisfied under these conditions.

The salient feature of this implementation is that the various subjects have effectively different "power" with respect to memo-objects, due to the different read-activators in their domains. That is why they can safely share the same set of tickets, contained in file dir, and still have different privileges. As we will see next, the situation is quite different in the access-control case.

Under the AC scheme, we assume that the operator read demands that the right "read" is in the operand-ticket. Since all subjects involved must have the right to invoke read, the difference between the subjects can only be in terms of the memo-tickets, each with the "read" right,

which are available to them. Thus, the desired policy can be established as follows:

For a given memo-object m , let:

$$Z(m) = \{z_1, \dots, z_j\} \subseteq X$$

be the set of boolean attributes satisfied by m . Let target(m) be the set of subjects S such that for each of them $Y(S) \subseteq Z(m)$. This is exactly the set of subjects which by our policy should be allowed to read m . Therefore, the ticket $(m:\text{memo};\text{read})$ should be available to these subjects and to none other. In order to establish such a distribution of tickets, we suppose that every subject S in our system has a directory file, $\text{dir}(S)$, which is readable only by him. Whenever a new memo-object m is created, a non-copyable ticket $(m:\text{memo};\text{read})$ should be stored in $\text{dir}(S)$ for every S in target(m), and in nowhere else. This is essentially the solution given by Jones and Wolf to a similar problem [Jon75].

Let us now compare these two implementations of our policy, along two dimensions: the number of control-object, which are needed for the implementation of the policy, and the complexity of the distribution of these objects.

As to the number of control objects, suppose that there are NS subjects in a system which are to be allowed to read memos, and let there be M memo-objects. Let K be the average number of subjects which are allowed to read a memo-object. The AC implementation requires $K \cdot M$ memo-tickets to be stored in the system, while under the OC-implementation only M tickets are required.

Even more important than the number of the control objects, is the complexity of their distribution. The AC-implementation requires a very specific distribution of the memo-tickets among the NS files dir(S). This distribution of tickets is itself a formidable task. Moreover, every file dir(S) must be well protected, and readable by the specified subject S only.

The situation under the OC implementation is much simpler: Once the NS different read-activators are correctly distributed among the various modules, we can store all the M tickets in one file, which is readable by everybody and does not have to be especially protected. This is obviously much less complex than under the AC-implementation.

References

- [Dah72] Dahl, O.J. and Hoare, C.A.R. "Hierarchical program structures," in Dahl, Dijkstra and Hoare, Structured Programming. New York: Academic Press, 1972.
- [Dij]76 Dijkstra, E.W. "A discipline of programming", Prentice Hall, 1976.
- [Jon75] Jones, A.K., Wulf, W.A., "Towards the design of secure systems", Software practice and experience 321-336 (1975).
- [Jon76] Jones, A.K. and Liskov, B., "An access control facility for programming languages", Carnegie Mellon U., Tech. Report, 1976.
- [Lis74] Liskov B. and Zilles S., "Programming with abstract data types." Sigplan notices, April 1974.
- [Min76a] Minsky, N., "Intentional resolution of privacy protection in database systems", Comm. ACM, March 1976.
- [Min76b] Minsky, N., "An activator-based protection scheme", Rutgers Tech. Rep., July 1976.
- [Mor73] Morris, J.H. Jr, "Protection in programming languages, CACM, Vol. 16, no. 1, (Jan. 1973).
- [Pal73] Palme, J., "Protected program mudules in Simula 67", Res. Inst. of Natl. Defense, Stockholm 80, Sweden.
- [Weg74] Wegbreit, B., "Procedure closure in El1", The computer Journal Vol. 17, no. 1, 1974.
- [Wei69] Weissman, C., "Security controls in the ADEPT-50 time-sharing system," in 1969 FJCC, AFIPS Conf. Proc., vol. 35, 119:133.
- [Wul74] Wulf, W.A., A note in SIGPLAN Notices.
- [Wul76] Wulf, W., London r.l., Shaw M., "Abstraction and verification in Alphard", CMU TR 1976.

To be published in the "Int. Journal of Computer and Information Sci."

SOSAP-TR-33

April 1977

AN OPERATION-CONTROL SCHEME FOR AUTHORIZATION IN COMPUTER SYSTEMS

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

ABSTRACT

The access-control authorization scheme, which is being used for the protection of operating systems, is found to be inadequate in other areas, such as in databases and information systems. A new authorization scheme, which is a natural extension of access-control, is proposed. The new scheme, which is called "operation-control", is shown to be superior to the access-control scheme in a number of ways. In particular: it facilitates more natural and efficient representations of policies, particularly the type of complex policies which appear in information systems; it facilitates enforcement by compile-time validation due to a greater stability of authority-states; and it reduces the need for revocation.

Key words and phrases: protection, access-control, operation-control, authorization, operating systems, information systems.

CR Categories: 4.29, 4.33, 4.35

TABLE OF CONTENTS

- 1: Introduction
- 2: The access-control (AC) approach to protection
- 3: The operation-control (OC) scheme
 - 3.1: Terminology and conventions
 - 3.2: Activators and the enforcement mechanism
 - 3.3: The activation patterns
 - 3.4: Control over the generation of objects
 - 3.5: The control-objects: their role and behavior
 - 3.6: The structure of domains and their dynamic behavior
 - 3.7: The global condition of activators
 - 3.8: On the kernel of the protection mechanism
- 4: Discussion
 - 4.1: Conceptual simplicity
 - 4.2: Expressive power
 - 4.3: Efficiency
- 5: Conclusion

1: INTRODUCTION

Authorization in computer systems is a discipline under which an action on the system can be carried out by a user, or by one of the modules of the system, only if the actor is authorized to perform this action. Such a discipline is necessary for the protection of the security and of the semantic integrity of systems.

Most current protection techniques are based on the so called access-control approach to authorization. This approach has been developed by Lampson [Lam71], Graham and Denning [gra72], Wulf and Jones [Wul74], and others, mostly in the context of operating systems, and it enjoys a considerable degree of success in this area. Unfortunately, however, this success has not been matched in other areas, such as databases and information-systems. It is our contention that this failure is due to some fundamental limitations of access-control as a scheme for representation of authority-structures. These limitations are discussed in section 2. A generalization of the access-control authorization scheme is suggested in section 3, and its merits are discussed in section 4.

2: THE ACCESS-CONTROL (AC) APPROACH TO AUTHORIZATION

The access-control approach to protection and authorization is well documented in the literature. In particular, the reader is referred to the excellent review articles by Saltzer [Sal75] and Linden [Lin76]. Here we will outline only the essential features of this approach, and we will discuss some of its limitations.

The system to be protected is formally viewed as a four-tuple (B, O, J, U) , where: B is a set of objects; O is a set of operators; J is a set of subjects, which are the actors that actually apply operators to objects, and are thus responsible for the dynamic behavior of the system; and U is the authority-state of the system. The authority-state is formally defined as a set $\{(S, b, o)\}$, where a triple (S, b, o) is the permission for subject S , which belongs to J , to apply operator o to object b . In other words, (S, b, o) is a permission for S to have "access o to object b ". Of course, the system must be supported by an enforcement mechanism which guarantees that only operations which are permitted by the authority-state U are carried out.

There are a number of ways to represent the set of permissions $\{(S, b, o)\}$. A method which is particularly relevant to this paper is called the capability-based-protection [Wul74, Lam76]. Under this version of the AC-scheme the authority-state of a system is represented by a distribution of special "control-objects" which we call tickets. A ticket is a pair $c=(b;r)$, where b is an identifier of an object, and r is a subset of a finite set of symbols, called "rights" (or access-rights), which identify, in some way, the operators that can be applied to object b . That is to say, the subject S who possesses the ticket $(b;r)$ is allowed to apply to b the operators identified by r . The generation and transport of tickets is tightly controlled so that the mere fact that a subject S has the ticket $c=(b;r)$ is taken as uncontested proof that S is authorized to have the specified access rights to object b .

Thus, under the AC-scheme (or, rather, under the capability-based^{*} version

(*) The phrase "capability-based" used for this version of the access-control protection is appropriate, even though we are using the term "ticket" for what is usually called "capability", because the set of tickets owned by a given subject determines his capabilities with respect to the system. (In this paper the term capability will be used in its colloquial sense).

of it) the tickets are used as the elementary building blocks of authority-structures, kind of "elementary-particles" of authority. Unfortunately, tickets are not suitable to serve as the only elementary particles of authority, for a number of reasons:

First, every ticket represents privileges with respect to a specific object, the one addressed by it. These privileges are independent of the value (or state) of the object. The problem is that authority structures are frequently based on the value of the objects involved, and are independent of their identity. To demonstrate the difficulty here, consider the following example. When a highway patrolman is sent to his duty he has to be given the authority to cite traffic violators. This authority is not given to him in the form of tickets, one for each violator. Indeed, the patrolman's authority cannot be defined in this form because at the time that the patrolman is sent to his duty the traffic violators do not exist, and the identity of the future violators is not known, so that it is impossible to construct individual tickets for the violators at that time. The point is that the patrolman's authority has to do with the behavior of motorists, not their identity. Tickets are too specific for this purpose, and at the same time, they are not sensitive enough, being independent of the properties (values) of the objects addressed by them.

Another problem with tickets is due to the unit of activity which they are designed to authorize. Every ticket represents a permission to apply certain operators to the object addressed by it. However, the activity of a subject may not be expressible purely in terms of operations on individual objects. One may have to use interactions between objects, where by "interaction" we mean an operation which involves several objects, and which cannot be decomposed into a sequence of legal unary operators on the individual objects. The problem is how

to express privileges with respect to an interaction purely by means of tickets, which represent permissions to perform operations on individual objects. When the interaction itself cannot be decomposed into such operations. (This difficulty will be demonstrated by an example in section 4.2.2).

Our conclusion from these observations is that there is a need for a new type of control object which can authorize directly interactions between objects, and which would not be based exclusively on the identity of the objects involved in operations. Such a control-object, which we call an activator, is the basis for the protection scheme proposed in this paper.

3: THE OPERATION-CONTROL (OC) SCHEME

The protection scheme to be introduced in this section is "capability-based" in the sense that it associates with every subject a set of control-objects which determine its capabilities. However, our scheme differs from the access-control scheme by the nature of these control-objects. In addition to the tickets which, as under the AC-scheme, represent privileges with respect to objects, we have control-objects called activators, which represent privileges with respect to operators, in the following sense: Every activator A identifies an n -ary operator q (for $n \geq 0$), specifying the conditions under which q can be invoked by the subject who possesses A . There may be several such activators for the same operator q , which may impose different pre-conditions on the activation of q , representing different privileges with respect to q . We are using the name operation control (OC) for this scheme because the activators possessed by a subject determine directly the type of operations which he can perform, not just the "accesses" that he has to various objects.

3.1: Terminology and Conventions

To set up the stage for our discussion we now introduce our interpretation for some well known terms such as object, subject, domain and type.

3.1.1 Objects and their types: We base our approach towards types on the type-concept used in Hydra [Jon75], which is briefly as follows:

The set of all objects in a system is described in terms of the three-level tree in Figure 1. The root of this tree is a primitive and unique object called "template". The objects at the second level are instances of this "template", and are called template-objects, or type-objects. Each of these type-objects, such as the object t, serves as a template for a set of objects in the third level, which are said to be "objects of type t", or "t-objects". A template t is supposed to contain the structural definition of all its instances.

* Fig 1 *

In order to impose some behavioral discipline on objects, we introduce the concept of "protected type". This is a type t for which there is a fixed set of operators (procedures) which have the exclusive ability to manipulate and observe t-objects directly. We will say that such an operator is "privileged with respect to t". The set of all these operators, for a given t, is denoted by privileged(t). Thus, a t-object for a protected type t can be manipulated either directly, from within one of the privileged(t) operators, or indirectly by invoking these operators. An important special case is an operator which is privileged with respect to only one type t: such an operator will be called a t-operator. (Note that the existence of a fixed set of t-operators is the

basis for the notion of "abstract-data-type" as it is defined in CLU [Lis74] (*)).

3.1.2 Sharable objects and their tickets: We distinguish between two broad classes of objects, to be called "sharable objects" and "concrete objects". A concrete object is one which is physically contained in one's "domain"(**). For example, the integer 7 and the symbol "seven" are concrete objects. A sharable object, on the other hand is not contained in any private domain, but it can be shared, or accessed, by several subjects, by means of a concrete object called ticket (which is essentially identical to the ticket of the access-control scheme).

With every type t of sharable-objects we associate a (possibly empty) set of symbols:

$$\text{rights}(t) = \{r_1, \dots, r_n\}$$

Each of these symbols, r_i , will be called a "right" with respect to type t .

(*) It should be pointed out that our scheme is not based on the concept of "privileged-operators". It is the other way around: we will see later that privileged operators can be implemented under our scheme. This concept is mentioned at this point to accommodate some of our examples in the following sections.

(**)The concept of domain will be defined later; for the moment it is enough to see it as the workspace of some user.

A ticket c for an object b of type t is defined to be a concrete object which will be denoted by:

$$c = (b:t; r)$$

where r , or $r(c)$, is a subset of rights(t). If a given right ri is in $r(c)$, we will say that "the ticket c has the right ri ". Since the content of a ticket depends on its t component we will use the phrase t -ticket to identify all tickets which address t -objects. A t -ticket which has all its possible rights will be denoted by $(b:t; ALL)$. Note that the symbols ri have been called "rights" in anticipation of their role in our protection scheme, which will be discussed later. (Note: the " t " part of c signifies that b is the identifier of an object of type t . Whenever the type of b can be understood from the context we will use the simplified notation $(b;r)$ for a ticket.)

In general, there may be several tickets which point to the same object b . We will say that such tickets are related. Let c_1, c_2 be two related tickets. We will say that c_1 is weaker, or less permissive, than c_2 if

$$r(c_1) \subseteq r(c_2)$$

Also, we will say that c_1 is strictly weaker than c_2 if

$$r(c_1) \subset r(c_2)$$

As to the generation and manipulation of tickets, the following is assumed. Tickets cannot be changed, and they can be generated only in the following two ways:

- a) When a new sharable object b of type t is created, a ticket for it is also created, with all of rights(t) in it. This ticket, $(b:t; ALL)$, will be called the primary ticket of object b .

- b) Given a ticket c it may be possible to generate a new ticket c' , which cannot be stronger than c . Such c' , which addresses the same object as c , will be called a derivative of c . (Later we will see when it is actually possible to generate derivatives of a given ticket).

3.1.3 The facade of objects: One of the objectives of our scheme is to enable value-dependent authorization. However, one cannot always expect to be able to base the decision as to the legality of an operation on the value of all the attributes of an object. This is due to the following reasons:

- a) It may happen that an attribute of an object is not directly observable. As an example consider the hidden components of "abstract-data-types".
- b) It may happen, in certain types of objects, that the mere act of observation of an attribute of an object introduces a change in the object itself. An example of this "quantum mechanical effect", is a record on a tape, which cannot be observed without repositioning the tape.
- c) One may not want to allow the use of certain attributes for protection, because such a use may itself reveal confidential information about the object.

For these, and other reasons which include efficiency, we now introduce the concept of the "facade" of an object, which is the part of an object which is usable for protection purposes. More formally, with every type t we associate the set:

facade(t)

which is the set of attributes of t-objects which are usable for authorization purposes. By convention, the facade of primitive scalar objects, such as real and integer numbers, is their value.

3.1.4. Subjects and their domains: A computing system changes in time in response to instructions submitted to a processor for execution*, where an "instruction" is a request to apply a specific operator to a specific sequence of operands. We will distinguish between two types of instruction sources:

- a) An external source, such as a human user sitting at the terminal.
- b) An internal source, which is a procedure maintained as an object in the system.

One important difference between these two types of instruction-sources is that an external source is totally unpredictable, as far as the system is concerned; while the behavior of a procedure can be at least partially predicted ahead of time.

We define a subject to be a pair

(INS,D)

where INS is an instruction-source, and D is the collection of objects which are directly addressable by INS. D will be called the domain of S. We will later see that the domain of a subject determines its capabilities, and also serves as its workspace.

*For simplicity, we assume that there is just one processor in the system.

Corresponding to the two types of instruction-sources, we distinguish between two types of subjects. A subject (INS,D) whose INS is an external source will be called a user, and a subject whose INS is an internal-source will be called an operator.

3.1.5 Operators: Operators are the dynamic components of a system. Every sequential process in the system can be described as a sequence of operations, each of which is the application of an operator to a sequence of zero or more operands. An operator may have various side effects on the system, but only one value which is called the outcome of the operator. The outcome is a concrete object which is stored in the domain of the subject which invokes the operator.

We will distinguish between two types of operators. First there is a fixed set of primitive operators whose internal activity would not be subject to the control of our protection mechanism. For example, the set of machine-instructions may be considered the primitive operators of an operating system. Secondly, an operator may be a subject (INS,D) whose source of instruction is a procedure maintained by the system. Note the recursive nature of the operator concept: A procedure, which is the INS component of some operator, has been defined to be a source of instructions, while an instruction is a request to invoke an operator.

3.1.6: Authorization scheme and policies: Following Jones and Wulf [Wul 74] we distinguish between the concept of "authorization-scheme" and that of "policy". A policy is a specific discipline which one would like to impose on a system. It will occasionally be called "authority-structure". An authorization (or protection) scheme is a framework which should be general enough to accommodate a variety of policies, as efficiently and conveniently as possible. Such a scheme

consists of two main components: A "language" which can be used for the specification of policies, and an enforcement-mechanism which guarantees that no illegal operations are carried out.

3.2: Activators and the enforcement mechanism

An activator, is a concrete object which we denote by:

$$A = \langle o, p_1, \dots, p_k \mid G \rangle \longrightarrow p_o$$

Here A is the name of the activator; o is an operator-identifier; p_i , for $i=1, \dots, k$ is a condition on the i -th operand of o , to be called "operand pattern"; G is a condition defined on all operands, and possibly on other objects in the system, (it will be called the "global condition" of A); and p_o is a condition on the outcome (result) of the operator, to be called the "outcome pattern". (Whenever we do not wish to distinguish between operand patterns and outcome patterns we will use the term "activation pattern" or just "pattern".)

The existence of an o -activator A in the domain of a subject S represents the authority for S to apply the operator o to any objects q_1, \dots, q_k in the domain of S , provided that for every $i=1, \dots, k$ the operand q_i matches the operand-pattern p_i of A (satisfies the condition p_i), and that the global condition G of A is satisfied. The activator A also gives S the authority to

 (*) If the operator o does not have an outcome then the part " $\longrightarrow p_o$ " will not appear in our notation. Also, the condition G may be absent. Thus, an activator may be denoted simply by $A = \langle o, p_1, \dots, p_k \rangle$

introduce into its own domain the outcome of the operator o , thus invoked, provided that this outcome "matches" the outcome-pattern po of A (that is, satisfies the condition po). To support this interpretation of the activators the following enforcement mechanism is proposed.

Let us define an instruction to be the construct

$$A(q_1, \dots, q_k)$$

where A is an o -activator for some operator o and q_i are its operands. It is also assumed that a subject can form such an instruction only from concrete objects A, q_1, \dots, q_k which exist in its own domain. Thus, the set of instructions which are expressible by a subject is directly determined by the content of his domain. Moreover, such an instruction is carried out only if the operands "match" the activation patterns as described above, and if G is satisfied. It is the responsibility of the enforcement mechanism to perform this "pattern matching" and to guarantee that no illegal operations are carried out. Once an operation is carried out, its outcome, if any, is checked. If it matches the pattern po it will be added to the operating-domain; otherwise, the value of the operation is lost, and an error procedure may be invoked.

Note that an operand q_i may be of two types: it may be a concrete object, such as an integer number, which stands for itself; or it may be a ticket that addresses a variable object, which is the real operand. Even in the latter case we will usually refer to q_i as an operand, relaying on the context to determine whether q_i itself is meant or the object addressed by it.

Thus, it is clear that the content of the domain D of a subject S, at a given moment in time, determines the set of operations which can be carried out by S at this moment. We can say therefore, that the domain of a subject determines its capabilities, or its authority.

There is an instructive analogy between the role of the activators in our scheme and the role of enzymes as the control devices of the living cell. The function of every enzyme is to facilitate a certain chemical reaction. Such a reaction takes place if there are enough substrates in the cell which fit the activation-sites on the enzyme, in some analogy to the function of our activator (see figure 2). Although this analogy between activators and enzymes should not be carried too far, it does provide an interesting viewpoint of the proposed scheme.

* Fig 2 *

Note the similarity between the activators and the formal-parameters-specification (or FPS) of procedures in programming languages: Both determine the legal set of operands of an operator. There is, however, an important difference between these two. Our activator is an independent object, disconnected from the operator which it activates. Moreover, while there is just one FPS per operator we will see below that there may be several different o-activators for the same operator o, which have different strength. The concept of "strength of activators" is defined below.

Let A be an activator of order k (with k operand-patterns). We define range(A) to be the set of all possible $(k+1)$ -tuples (q_1, \dots, q_k, q_0) of objects, which can be matched with the corresponding activation-patterns of A , and which satisfy the condition G of A .

Let A and A' be two o -activators for a given operator o . We will say that A' is weaker than A (or, equivalently, A is stronger than A') iff

$$\text{range}(A') \subseteq \text{range}(A).$$

We will say that A' is strictly weaker than A iff

$$\text{range}(A') \subset \text{range}(A),$$

such an A' will also be called a reduction of A .

As to the generation and manipulation of activators, the following is assumed: First, there is no way to change an existing activator, except to erase it. Secondly, new activators can be generated only in the following two ways:

- a) When a new operator o is created, an o -activator is generated with it. It will be called the primary o -activator.
- b) Given an o -activator A , it may be possible to generate a new activator A' , which is called a derivative of A . A' cannot be stronger than A . (Later we will see when it is actually possible to generate such a derivative.)

The following properties of the activators follow immediately from the above:

- a) The set of all o -activators, for a given operator o , is partially ordered with respect to the relation "stronger".
- b) Every activator is stronger than all its derivatives

- e) The primary o-activator is the strongest o-activator.

3.3: The activation patterns

To be more concrete about the activators and their use we have to suggest a specific structure for the activation-patterns. The structure to be described below is designed to support many of the known authority structures in computer systems. Note that the run time overhead due to the enforcement mechanism which is necessary to support our scheme depends on the complexity of the activation patterns and that of G. In this paper we do not impose any restriction on this complexity, because such restrictions should depend on the nature of the system to be protected.

3.3.1: Operand-patterns: An operand pattern P, to be denoted by

$$[I;R;V]$$

is a conjunction of three predicates I,R,V, which are called components, or sub-patterns of P. They are defined below.

The sub-pattern I (which is the only mandatory part of P) is called the identity-based sub-pattern. It is either a type identifier, "t", which is meant to be satisfied by any object of type t; or it is the phrase "b:t" which is satisfied only by the particular object b of type t. The entire pattern P whose I component identifies a type t will be called a t-pattern. The structure of the two other components of a t-pattern depends on t. If a subpattern R or V does not appear in P, it would be interpreted as identically TRUE, which means that it does not impose any restrictions on the object matched to the pattern.

The sub-pattern R , called the privilege-based subpattern, is applicable only in the case that t is a shared-type. R has the general form:

$$R = r_1 \& r_2 \& \dots \& r_k$$

where each r_i is a symbol which belongs to rights(t). R is meant to be satisfied by any ticket of a t -object which contains at least the right r_1, \dots, r_k .

The sub-pattern V , called the value-based subpattern, is a predicate defined on the facade of the object being matched with it.

An example: Let doc be a type of sharable objects which carry documents in a military information system. Let the facade of doc-objects be defined by:

$$\text{facade}(\text{doc}) = \{\text{slevel:integer, category:text}\}$$

where slevel is an integer which specifies the "security-level" of the document, and category specifies its category, such as "navy" or "army". These two attributes are the traditional security parameters in military establishments [Wei69]. Let

$$\text{rights}(\text{doc}) = \{U, E\}$$

As we will see below, the symbols "U" and "E" stand for the rights to update and erase a document, respectively.

Suppose now that there are three doc-operators: read, update and erase, which are the only operators able to manipulate a document directly (see section 3.1.1). The primary activators of these operators are as follows:

READ = <read, [doc] >

UPDATE = <update, [doc;U], [text] >

ERASE = <erase, [doc;E] >

The activator READ can be applied to any doc-ticket, displaying the content of the document. The activator UPDATE can be applied to a doc-ticket which contains the "U" right. The second operand of UPDATE, which can be any text-object, specifies the nature of the desired update. The activator ERASE can be applied to any doc-ticket which contains the "E" right, erasing the content of the document.

The right "U" can properly be considered an "update-right" due to the following reason: "U" is required by the primary update-activator, which means that it would be required by all update-activators. Thus, the update operator can never be applied to a doc-ticket which does not have the "U" right. A similar argument would show that "E" is the "erase-right".

As has already been explained, the primary o-activator, for any given operator o, allows for the most general use of o. In order to provide for a more limited use of o one creates weaker derivatives of the o-activator. For example, the activator

ERASE' = <erase, [doc;E,U] >

is weaker than ERASE because it can be applied only to a doc-ticket which has both "U" and "E" rights in it. The following activator

$$\text{ERASE}'' = \langle \text{erase}, [d:\text{doc}; E] \rangle$$

is also weaker than ERASE, because it can erase only a specific document d . Note, however, that there is no ordering relation between ERASE' and ERASE''. Neither of them can be a derivative of the other.

To illustrate the use of value-based sub-patterns consider a subject S whose domain D contains the following activators.

$$\text{READ}' = \langle \text{read}, [\text{doc};; \text{slevel} \leq 2] \rangle$$

$$\text{UPDATE}' = \langle \text{update}, [\text{doc}; U; \text{slevel} \leq 2 \ \& \ \text{category} = \text{"navy"}], [\text{text}] \rangle$$

which are reduced derivatives of READ and UPDATE, respectively. S has the power to read any document whose security level is smaller than or equal to 2 and whose ticket he can get. S can also update "navy" documents whose $\text{slevel} \leq 2$, provided that he has a ticket with the "U" right for such a document. However, S cannot erase any document because he does not have any erase-activators.

3.3.2 The outcome-pattern: The outcome-pattern po of an activator

$$A = \langle o, \dots \rangle \longrightarrow po$$

is a condition on the outcome of the operator o , when invoked by means of A . This means that only an outcome which satisfies po can be added to the operating domain by using A . The structure of the outcome-patterns is identical to that

of the operand-patterns. However, the interpretation of the R-component of the pattern is different. Let p_0 be the pattern $[I; r_1 \& r_2 \& \dots \& r_n; V]$. The right-symbols r_1, \dots, r_n in this pattern are not treated as conditions on the rights $r(c)$ contained in the ticket c returned as the outcome of the operation. Rather, they serve as a filter on $r(c)$, in the following sense: Any right in $r(c)$ which is not represented in r_1, \dots, r_n would be erased from the outcoming ticket c . Thus, the component R of p_0 serves as the upper limit for the rights which might be returned as a result of applying the activator. This means, for example, that the activator

$$\langle o, p_1 \rangle \longrightarrow [I_1; r_1; V_1].$$

is weaker than

$$\langle o, p_1 \rangle \longrightarrow [I_1; r_1, r_2; V_1]$$

Returning to our document example, consider an operator getdoc which retrieves documents from files. Let the primary activator of getdoc be

$$GET = \langle \text{getdoc}, [\text{file}], [\text{text}] \rangle \longrightarrow [\text{doc}; \text{ALL}]$$

The first operand of getdoc must be a file of documents, in which getdoc is supposed to locate a document identified by the second parameter, returning the ticket for the document as its outcome. Note that the outcome-pattern $[\text{doc}; \text{ALL}]$ would match any document-ticket. Consider now the following, weaker, derivative of GET:

$$GET' = \langle \text{getdoc}, [f_1: \text{file}], [\text{text}] \rangle \longrightarrow [\text{doc}; U; \text{slevel}=1]$$

GET' can get documents only from a specific file f1; moreover it can produce only tickets for documents whose security-level equals 1, and these tickets can have no more than the "U" right in them.

3.4 Control over the generation of objects

As we saw in the last section, one can control the use of individual existing sharable objects by the distribution of their tickets. We will now show how the generation of new objects can be controlled. (Only the essentials of such control are discussed, leaving some details unspecified). This will serve as a farther illustration of activators and their patterns.

First, we assume that there is a primitive operator gen-type which is able to generate new type-objects (the objects in the second level of the tree in Figure 1). Let the primary activator of this operator be:

$$\text{GEN-TYPE} = \langle \text{gen-type}, \dots \rangle \longrightarrow [\text{template}; \text{ALJ.}]$$

This activator has a sequence of operand-patterns not specified here, which determine the type of arguments which are required by gen-type. Invocation of GEN-TYPE would generate a template-object returning a ticket for it with all its possible rights. Obviously, only a subject who has the GEN-TYPE activator, or some derivative of it, can generate new types.

We now assume that together with a new type-object t, the following "instantiation-activator" is generated

$$\langle \text{gen-t}, p_1, \dots, p_k \rangle \longrightarrow [t; \text{All}],$$

where gen-t is an operator which generates instances of type t, and p1,...,pk determine the arguments required by gen-t. Application of this activator to an appropriate sequence of operands returns a ticket to the newly formed t-object, with all the rights(t) in it. For example, the primitive instantiation-activator for the type doc may be:

GEN-DOC = <gen-doc, content:[text], slevel:[integer],
category:[text] > → [doc;ALL]

(To distinguish between the various operand-patterns we use labels such as "content:[...]"). The three operands of gen-doc determine the initial state of the generated document: its content, security-level and category. A subject who has this activator can generate documents with arbitrary security-level and category, and he gets a ticket for the generated object with all its possible rights. However, a subject who has the following derivative of GEN-DOC

GEN-DOC' = <gen-doc,...,category:[text;;value="navy"]> → [doc;E]

can generate only documents whose category is "navy", getting for them tickets without the "U" right. Note that documents generated by GEN-DOC' can never be changed, because there can be no tickets with the "U" right for them.

As to the control over the distribution of the instantiation-activators, the following is assumed: The primary instantiation activator of a type t is stored inside the template-object t itself. It can be accessed, and copied or moved to other places by a subject who has an access to an appropriate ticket of the template object t. (See section 3.8 for information about the transport of activators and tickets.)

3.5 The two types of control-objects: their role and behavior

Our protection scheme is based on two primitive types of objects, activators and tickets, which we call, collectively, "control-objects". The distribution of control-objects throughout the system serves to determine its authority-state, namely, such distribution determines who can do what in the system. The roles of the two types of control-objects is reviewed in this section, and their transport is discussed.

There is a symmetry in the function of activators and tickets in our scheme. A ticket for object b which resides in the domain D of a subject S represents the privileges that S has to b , in the sense that the ticket determines the set of operators which may be applied by S to b . Analogously, an o -activator which resides in D represents the privileges that S has to the operator o , in the sense that it defines the set of objects to which o can be applied by S . Tickets and activators also play complementary roles in our scheme: neither one of them alone is sufficient for the application of an operator to a sharable-object. For this, one needs both an activator and a ticket (or several tickets) which fit the activator.

The complementarity of activators and tickets allows us to formalize the semantics of the right-symbols. Let the symbol r_1 belong to rights(t) for a given type t . We define the privileges associated with r_1 to be the set of t -patterns of the various activators in the system which require r_1 . Moreover, for a given domain D we define the local privileges associated with r_1 to be the set of t -patterns in D which require r_1 . Note, for example, that this set may be empty, rendering r_1 useless in the context of D , even if the set of global privileges of r_1 is non-empty. For instance, the right "U" of section 3.1 would be useless within a domain which has no update-activators.

The two types of control-objects exhibit some similar structural and behavioral characteristics, which are best seen by comparing the following two sets: the set $T(b)$ of all tickets for a given object b , and the set $A(o)$ of all o -activators. $T(b)$ and $A(o)$ are partially ordered sets with respect to the relation stronger, defined for tickets and activators respectively. Every ticket in $T(b)$ is a direct, or indirect, derivative of the primary ticket of b , which is created together with the object b itself. Likewise, every activator in $A(o)$ is a derivative of the primary o -activator, which is created together with the operator o . A control-object, whether it is an activator or a ticket, is stronger than all its derivatives.

Control-objects are to be distributed by means of two primitive "transport-operators": k-copy and k-move. ("k" stands for "kernel", as these operators should belong to the kernel of the system, to be discussed in section 3.8). Each of these operators when applied to a control-object co generates a new control-object co' in some other place in the system, such a co' cannot be stronger than co. The difference between the two transport operators is that k-copy does not affect the original control-object while k-move erases it. Thus, k-move, in effect, moves a control-object from one place to another, possibly reducing it in the process.

In order to get a degree of control over the transportability of individual control-objects, the following facility is introduced. The facade of a control-object of either type, consists of two boolean components, "copy" and "move", to be called the "intrinsic rights" of the object. The operator k-copy can be applied to a control-object co only if "it has the copy right", namely, if the "copy" component of co is TRUE. Likewise, k-move can be applied only to a control object which has the "move" right. Thus, a control-object with

neither intrinsic-rights is untransportable. Of course, the control-object co' generated from co by one of these operators cannot have more intrinsic-rights than co , but it can have less.

It is obviously vital to have some control over the use of the transport-operators. Such a control can be achieved by the distribution of their activators. These activators will be discussed in section 3.8.

3.6 The structure of domains, and their dynamic behavior

As has already been explained, the content of a domain D at a given moment in time, T , determines the set of operations which can be carried out at time T by the subject S associated with D . But what can we say about the future capabilities of the subject S ? To answer this question one must be able to predict the future content of D . This in turn, requires an understanding of the dynamic behavior of domains.

3.6.1 External and internal changes of domains. We will distinguish between two types of domain change, to be called "external changes" and "internal changes". An external change of a domain D associated with subject S is a change caused by an operation invoked by another subject, S' . In particular, it is such a subject S' which created D in the first place. In order to predict the dynamic behavior of a given domain D under external changes one must be able to tell which subjects have the capability of changing D , and what are they up to.

An internal change of a domain D is one which is caused by an operation invoked by its own subject S , as follows: Let

$$A = \langle \dots \rangle \rightarrow p_0$$

be an activator in D, and let $A(q_1, \dots, q_k)$ be an operation invoked by S. The outcome of this operation, if any, is added to D. This outcome is a concrete object which satisfies p_0 . (Note that depending on p_0 , the outcome which is added to the domain may be a primitive object such as integer, a ticket for a sharable object, or even an activator). Thus, the nature of the possible internal changes of a domain D are determined explicitly by the content of D itself.

Now, it seems reasonable to assume that in a well-designed system there would usually be only a small number of subjects S' which are able to change an existing domain D. Moreover, even these subjects are not likely to exercise their ability to change D very often, so that an external change of a domain is likely to be a relatively rare event. Therefore, we continue our discussion of the dynamic behavior of domains taking only internal domain-changes into account.

3.6.2 The structure of domains: Until now, domains have been presented as monolithic structures. We will now distinguish between two parts of a domain, to be called "permanent" and "transient" parts. The permanent-part of the domain of a subject S is created with the subject, and is attached to it throughout its lifetime. We will sometimes refer to this part simply as "the domain" of the subject. The transient-part of the domain, to be also referred to as the "workspace" of the subject, exists only for the duration of a single "activity-period" of the subject. In the case of a user an activity-period is a single "session" of user-system interaction. An empty workspace is attached to the domain of the user at the beginning of a session, only to disappear when the session terminates. An activity-period of an operator is the period between its

invocation and its return. When an operator is invoked, a new workspace, which contains all the operands, is attached to its domain, to be deleted when the operator returns control.

We now introduce the convention that unless specified otherwise, an internal-change of a domain effects its transient part only. This means that the outcome of an operation is usually stored, in the workspace of the subject, leaving the permanent part of the domain invariant of the activity of its own subject. An example will clarify all that.

3.6.3 An example: Consider a subject S whose domain, or rather, the permanent-part of its domain, is given in Figure 3. This domain contains two file-tickets, for files f1 and f2, which are assumed to contain documents. The domain also contains five activators: The activators GET' and GET" are reductions of the activator GET (cf. section 3.3.1). Each activator gets a doc-ticket identified by its second operand, from the file identified by its first operand. GET' can be applied only to a ticket of one file, f1. Namely, GET' can be used to generate tickets, with the "U" right, for documents stored in f1. We will denote the set of all such tickets by F1. The activator GET", which can be applied to the ticket c2 of the file f2, can generate a set, F2, of tickets of documents stored in f2 whose slevel=1. These tickets would have only the "E" right in them.

capig 3 *

The activators READ', UPDATE' and ERASE, already discussed in section 3.3, can be applied to the doc-tickets generated by GET' and GET". READ' can be applied to any document whose alevel \leq 2. Note, therefore, that some of the documents whose tickets may be generated by GET' cannot be read by S. UPDATE' can be used to update any "navy" document, provided that its ticket has the "U" right. This means that none of the documents from f2 can be updated by S, and only some of the documents on f1, the "navy" documents, can be updated. Finally, using ERASE, S can erase all the documents he gets from f1, but none from f2. (Note that our subject cannot generate new documents, because he does not have a gen-doc activator).

Note that all the doc-tickets generated by our subject would be inserted into its "workspace", which is the transient-part of the domain that disappears at the end of the session. That is to say, the subject S cannot have any doc-tickets for extended periods of time. This property of our scheme is very important as it reduces the need for revocation.

A comment: The domain in Figure 3 is incomplete in the sense that it contains no activators for basic operations such as integer-addition or manipulations of text variables. Such activators are necessary because, by our definitions, no operation can be carried out by subject without having the proper activator in his domain. However, following [Min76], we assume that all control-objects which are necessary to authorize the use of operators and objects which we do not wish to restrict are included, by default, in all domains.

3.7: The global condition of activators

We define the condition G of activators by the following two properties:

- a) G is a conjunction of predicates: $g_1 \& g_2 \& \dots \& g_k$
- b) The reduction of a global condition can be performed by adding a conjunct to it, not by a change of existing predicates.

At this point, no restrictions are imposed on the individual predicates g_i . In particular, g_i can be defined on all the operands of the activators, as well as on other objects in the system which are not otherwise involved in the operation. Moreover, we will allow g_i to have side effects. Here are some applications of the global-condition.

3.7.1: Correlation between operands: The operand pattern, p_i , has been defined to be a condition on the i -th operand. Since G can be defined on all operands, it can correlate them. For example, let copyd be an operator which copies the content of one document (doc-object) into another. Consider the following copyd-activator

$$\langle \text{copyd}, d1: [\text{doc}], d2: [\text{doc}; u] \mid d1.\text{category} = d2.\text{category} \& \\ d1.\text{slevel} \leq d2.\text{slevel} \rangle$$

The only restriction imposed by this activator on the individual operands is that $d2$ must have the "U" right, (without which the update of $d2$ would not be possible). However, the G part of the above activator requires the two operands to be of the same category, and that the second operand should not have a lower security level than the first. (This is a very common type of restriction in military information systems.)

3.7.2: Conditions on global status-variables: Suppose, for example, that there

is a global variable T in the system which represents the real-time. The following activator

$$\langle \dots \mid t_1 \leq T \leq t_2 \rangle$$

can be used only in the time period (t_1, t_2) because its G part would return **FALSE** at any other time. In a similar way one can construct activators which are conditioned on other global variables in the system.

3.7.3 Self destructive activators: Consider a predicate count-down of the form

```
BEGIN  OWN N;
      N ← N-1;
      IF N ≤ 0 RETURN FALSE;
END
```

If this predicate is used as a component of the G part of an activator A , it would limit the number of times that A can be used. If, for example, the own variable N of such a count-down predicate is initialized to 2 when the activator is created, then after two applications of A it will return false, preventing further use of A .

3.7.4: Revocation of activators^(*): One of the classical problems in capability-based protection is how to revoke a privilege already granted.

(*)I am indebted to Dorothy Denning for suggesting this important application of the global condition of activators.

Revocation of tickets has been studied extensively by Redell [Red74], Cohen [Coh75] and others. Here we will see how activators can be revoked.

Consider a subject S1 who has an activator

$$A1 = \langle \dots | g \rangle$$

in his domain D1. Let a_1 be a boolean variable local to D1. Suppose that S1 generates a derivative A2 of A1

$$A2 = \langle \dots | g \ \& \ a_1 \rangle$$

storing it in the domain D2 of subject S2 (see figure 4). It is quite obvious that A2 can be used only as long as the boolean variable a_1 is TRUE. Thus, although A2 belongs physically to S2, it is still controlled by S1 which can prevent the activation of A2 simply by turning off the variable a_1 . Moreover, every derivative of A2 would be controlled by S1 in the same way because it is impossible to remove a conjunct from G. Moreover, one can add additional controls in a similar way. For example, let a_2 be a boolean variable in D2, and suppose that S2 generates a derivative

$$A3 = \langle \dots | g \ \& \ a_1 \ \& \ a_2 \rangle$$

of A2, storing it in the domain D3 of S3 (see figure 4). Now, S1 can deactivate and reactivate both A2 and A3 by turning a_1 off and on, while S2 can control A3 in a similar way by means of its own variable a_2 .

 • Fig 4 •

Note that in a similar way one can construct a variety of revocation patterns. In particular, the variables a1 and a2 above may be stored inside some shared object whose tickets are distributed in a certain way.

3.7.5 Monitoring the use of activators: Using the ability of G to produce side effects, one can monitor the use of activators, as follows. Suppose that a subject S1 has the activator

$$A1 = \langle \dots | g \rangle$$

in his domain D1. Let

$$A2 = \langle \dots | g \ \& \ m \rangle$$

be a derivative of A1 where m is a predicate which always returns TRUE, and which is programmed to write a record into a file accessible to S1, reporting about each invocation in which it participates. Thus, S1 would have an audit trail of all activations of A2 and of any derivative of it, because m cannot be removed from an activator. The users of A2, or of its derivatives, may not be aware of such audit trail being formed, and they certainly cannot do anything about it, because no part of G can be removed from an activator.

3.8: On the kernel of the protection mechanism (*)

*) This section can be skipped on first reading.

The purpose of this section is to clarify and support some of the assumptions made in previous sections. In particular, it has been assumed that for every type t there is a set of privileged operators with respect to t , which have the exclusive ability to modify and observe t -objects directly. Here we will show how this exclusiveness can be imposed by means of the basic protection mechanism. This discussion will bring us down to the foundations of the protection mechanism, which is frequently called its kernel. However, only some aspects of such a kernel will be discussed here. Its complete study is beyond the scope of this paper because the kernel is likely to be strongly dependent on its context. For example, the kernel would surely be very different in the case of operating systems, databases, and programming languages. Therefore, the following discussion should not be viewed as a proposal for a specific implementation.

3.8.1: Segments and Their Operators: In an attempt to find a uniform implementation for all types of sharable objects, we first define a primitive type called segment which is to serve as a host for sharable objects of all types.

A segment is essentially a chunk of storage divided into a sequence of slots each of which can host one concrete object of a given primitive type. For example, there may be integer-slots, text-slots, ticket-slots and activator-slots. The various slots in a segment will be addressed by their relative position with respect its origin. This division of a segment into slots will be called the structure of the segment. A segment is allocated, to host an object of a given type t , by the gen- t operator (defined in Section 3.4). The structure of this segment is determined by t , and is fixed for the life-time of the object.

We will treat the set of all segments as a type. It is a special kind of type as it includes (*) all other types of sharable objects, since, by our definition, an object of any type is also a segment. As any other type of sharable objects, the type "segment" has its own rights, which, following Hydra, will be called "kernel rights". We assume that

$$\text{rights}(\text{segments}) = \{\text{k-read}, \text{k-write}\}.$$

A ticket (b:t; r) of a sharable object b can be viewed also as a ticket for the segment which hosts an object b, provided that we generalize the r-component of a t-ticket as follows:

$$r \in \text{rights}(t) \cup \text{rights}(\text{segment}).$$

That is to say, the kernel-rights are common to all types, and may appear in any ticket.

We now introduce two operators which operate on segments: the already mentioned transport-operators, k-copy and k-move. The primary activator of k-copy is:

$$\text{K-COPY} = \langle \text{k-copy}, s1:[\text{segment}; \text{k-read}], loc1:[\text{integer}], \\ s2:[\text{segment}; \text{k-write}], loc2:[\text{integer}], \text{reduction}:[\text{text}] \rangle$$

The operator k-copy copies a concrete object from slot loc1 of segment s1 into slot loc2 of s2. The fifth parameter, reduction, specifies the reduction to be

(*)The concept of "type-inclusion" could have been introduced formally in section 3.1.1. We avoided this for the sake of simplicity, and we are using type-inclusion, in an ad-hock manner, only in this case.

applied to the copied object, if it happens to be a ticket or an activator. Namely, it specifies in what way the new object is weaker than the original (The syntax of the reduction specification will not be discussed here). The conditions for the copy operation to be carried out, are of two types: explicit and implicit. The conditions which are explicit in the activator are that the ticket for s1 must have the "k-read" right, and the ticket for s2 must have the "k-write" right. In addition to these explicit conditions we assume that the following restrictions are built into the operator itself.

- 1) The concrete-object in slot loc1 of s1 must fit the type of slot s2 of s2. This simply means that the k-copy operator does not violate type specification of the structure of segments. Thus, only a ticket can be stored in a ticket-slot, only an activator can be stored in an activator-slot, etc. Moreover, a ticket-slot may be earmarked for ticket of a certain type of objects only and there may be a limit imposed on the rights which may be contained in the slot. All such specifications must be honored by k-copy.
- 2) If the content of the loc1-slot of s1 is a ticket or an activator, it must have the intrinsic-right "copy".

The second transport-operator, k-move, has a similar activator, K-MOVE. The only differences between these two operators are:

- a) k-move erases the content of slot loc1 of s1.
- b) In the equivalent of restriction (2) above, the intrinsic-right "move" (rather than "copy") is required.

Note that because of (1) above, a slot behaves like a variable in a typed language, namely, it has a specified range of objects which may be stored in it. A particularly important case is that of a "ticket-variable" which can contain tickets only for a given type of objects, and with a specified maximal set of

rights. Such a facility, which is similar to the ticket variables introduced by Jones and Liskov in [Jon76], can contribute greatly to proving correctness of policies.

As we will see next, the activators K-COPY and K-MOVE are too powerful to be contained in any domain. We will have to use restricted derivatives of them.

3.8.2. On the Implementation of Privileged-Operators:

For a given type t consider the following derivative of K-COPY:

READ- t = $\langle k\text{-copy}, s1:[t;k\text{-read}], loc1:[integer],$
 $s2:[myself], loc2:[integer], reduction:[text] \rangle$

This is a reduction of K-COPY in two ways. First, the $s1$ pattern can be matched only to tickets of t -objects (which is a smaller set than all segments, which are allowed by the pattern $s1$ in $k\text{-copy}$). Secondly, the phrase "myself", in the $s2$ pattern, is a context dependent pattern which matches only the operating-domain. This allows a subject which has this activator in its domain, together with a t -ticket:

$c = (b:t; k\text{-read}, \dots),$

to copy information from object b addressed by c , into its own domain. Or, in other words, READ- t allows reading of t -objects. In a similar way, the activator

WRITE- t = $\langle k\text{-copy}, s1:[myself], \dots, s2:[t;k\text{-write}], \dots \rangle$

allows one to copy information from the operating domain into t -objects, that is, to write into t -objects. One can also construct a pair of similar derivatives of K-MOVE, which move data into, and from t -objects.

We thus have four activators for a given type t which can be used for reading from, and writing into t -objects. Assuming that the primary activators of k -copy and k -move are not themselves available, only a subject which has in its domain one of these four activators is able to manipulate or observe directly t -objects. Thus, an operator is privileged with respect to t if and only if it has at least one of the above four activators in its own domain. Note that an operator can be privileged with respect to a number of different types. This is not possible under the implementation of "abstract-types" in CLU, for example.

4: DISCUSSION

The purpose of this section is to evaluate the proposed operation-control (OC) scheme along a number of dimensions, such as conceptual adequacy and simplicity, efficiency, and expressive power. To keep the discussion in perspective we will compare our scheme with the capability-based access-control (AC) scheme, specifically the Hydra [JON73, WUL74, COH75] version of it. Such comparison is appropriate because, as we will see next, the OC scheme can be viewed as a natural extension of the capability-based version of the AC scheme.

4.1: Conceptual simplicity

In comparing our scheme to the access-control one it may seem that we are sacrificing a great deal of conceptual parsimony by adding another type of control-object. This, however, is not the case. We will show next that the proposed scheme can be viewed as a natural extension of the AC-scheme, which results from the removal of an unwarranted restriction in it.

Note that the enforcement mechanism which is necessary to support the OC scheme is essentially identical to that of the AC scheme. Under the AC scheme, an operation $o(q_1, \dots, q_n)$ is considered legal if the tickets of the operands q_1, \dots, q_n satisfy the requirements imposed by the formal-parameter specification (FPS) part of the operator o (this part is called "template" in Hydra). Thus, the FPS of an operator is functionally equivalent to our primary-activator. Moreover, having the right to call an operator o , under the AC scheme, is equivalent to having the primary o -activator, under the OC scheme. Thus, the OC scheme can be viewed essentially as a AC scheme with the additional degree of freedom which allows the formation of a whole set of o -activators of different strength. These activators represent varying degrees of authority with respect to the operator o , just as the set of tickets of a given object b represent varying degrees of authority with respect to b . This symmetry in the treatment of objects and operators, which does not exist under the AC scheme, is important because it reflects a common feature of authority-structures. The capabilities of an actor (subject) in computer systems, as well as in the real world, is frequently due to the type of operations which he can perform, not only to the privileges that he has with respect to specific objects. Thus, our scheme is conceptually cleaner and more complete than the AC scheme.

4.2: Expressive power

We will say that a policy is expressible in a given scheme if it can be specified and enforced by means of the formal devices provided by the scheme. (Note that expressibility, so defined, is a stronger concept than implementability. Indeed, any policy can be implemented on any kind of system, simply by programming it into an interpreter which carries out every operation on the system.) The difference in expressive power between the AC and the OC

schemes is primarily along two dimensions: value-dependency and the ability to handle interactions. It is not that value dependent policies and policies with respect to interactions cannot be expressed in the AC scheme (although this is true for some such policies). The main problem is that the implementation of such policies tends to be cumbersome and inefficient to the point of being impractical.

4.2.1. Value-Dependent policies: We define a "value-dependent policy" to be one under which the legality of an operation depends on the value (or state) of the operands. In particular, we will be interested in the case where the value-dependency itself depends on the subject which invokes the operation. Such policies can be represented in our scheme by the distribution of activators with appropriate value-based patterns. On the other hand, the only way to represent such a policy under the AC scheme, is by a suitable value-dependent distribution of tickets. That is to say, the placement of tickets depends on the values of the objects addressed by them. As we will demonstrate later by an example, such a representation may be so costly and error-prone that it becomes completely impractical.

4.2.2 Handling of interactions: The concept of "interaction", mentioned in section 2, is defined, more rigorously as follows:

For a given subject S, an interaction is an n-ary operator, for $n > 1$, which cannot be expressed by S as a sequence of legal unary operations on its individual operands.

Note that by this definition an operator which is an interaction for one subject may not be an interaction for another. Consider, for example, a procedure appoint(e,j) which appoints employee e to job j in a corporate information system. Suppose that this appointment is actually done by planting a pointer to

i in the record which represents e, and vice versa. Suppose also that such tinkering with pointers is not allowed outside of the procedure appoint, for obvious reasons. Thus, for any subject other than the procedure appoint itself, the operator appoint(e,i) is an interaction because there is no way to decompose it into a sequence of (more primitive) operations on e and i separately.

Now, consider the following policy with respect to the interaction appoint. Let E1 and E2 be two sets of employees and let J1, J2 be sets of jobs. Let S be a subject who is to be allowed to appoint employees in E1 to jobs in J1 and those in E2 to jobs in J2, but is not allowed to appoint employees in E1 to jobs in J2, etc.

Under the OC-scheme, let pe1, pe2, pj1, pj2 be patterns which match members of the sets E1, E2, J1, J2 respectively. The desired policy is realized by giving S the activators <appoint, pe1, pj1 >, and <appoint, pe2, pj2 >.

To see the difficulty under the AC-scheme we now consider several attempts to represent this policy. First, suppose that the operator appoint requires the right r1 from its first argument and r2 from the second. Now, S can be given a ticket with r1 for every member of E1,E2; and a ticket with r2 for every member of J1 and J2. The problem is that this would allow S to make cross appointments, of members of E1 to jobs in J2, etc.

The desired policy can be implemented in a system such as Hydra, as follows: One may maintain a table inside the operator appoint, which identifies the set of triples (S,e,j) such that S is allowed to appoint e to job i. The operator appoint may be programmed to obey such specifications. However this is not a representation in terms of the AC-scheme, and, it is quite contrary to its underlying philosophy of the capability-based approach to authorization.

A correct but unnatural and very inefficient representation of our policy in terms of the AC-scheme is the following: For every "appointable" pair (e, j) we create an object e_j which represents the pair. The operator appoint can be considered as a unary operator on such pair-objects. The authority of our subject can be defined by giving him a ticket for every pair-object e_j which S is allowed to appoint. Such representation of authority apart of being highly artificial and inconvenient, may be extremely inefficient. For example, if the cardinality of each of the sets E_1, E_2, J_1, J_2 is N , then S will have to maintain $2N^2$ tickets, as opposed to two activators which are necessary under our scheme. Moreover, if the membership of an employee e in one of the sets E_1, E_2 , is determined by the value of e , then the maintenance of the authority-structure is very difficult. Whenever e stops to be a member of E_1 , say, one has to revoke all existing tickets for appointable pairs e_j .

4.3: Efficiency

The efficiency of a protection scheme should be measured along two dimensions: efficiency of the representation of policies, and the efficiency of enforcement. To explain what we mean by efficiency of representation we now introduce a number of concepts:

We will use the term "control-material" for the overall distribution of control objects throughout a system. For a given policy P , we will be interested in the following properties of the control-material which is necessary for the representation of P .

- a) The volume of the control material, which is the number of control objects in it.
- b) The complexity of the distribution of the control material. We will say that the distribution of the control material is more complex, if there

are more rigorous requirements as to the placement of the various control objects.

- e) The volatility of the control material. By the term "volatility" we mean, loosely speaking, the amount of change in the control material which is necessary in order to maintain a given policy during normal operation of the system, or to support incremental changes in the policy itself. (The term "stability" will also be used, as the opposite of "volatility").

These aspects of a protection scheme, and their relevance to the efficiency of the representation of policies, are discussed in the next two sections. Efficiency of enforcement is discussed in section 4.3.3.

4.3.1 The volume and the complexity of the control-material: In this section we will argue that

the volume of the control-material which is necessary for the implementation of a given policy under the OC scheme tends to be smaller, and less complexly distributed than under the AC-scheme.

An important reason for this tendency is that activators can have various degrees of generality (or, specificity), which can be adapted to the nature of the policy at hand. For example, the activator <read, [doc] > can be used to read any document, while the activator <read,[d:doc] > can be used only to read the specific document d. Tickets on the other hand have fixed degree of specificity: every ticket represents privileges with respect to one specific object. Thus one may need many tickets to represent a capability which, under the OC scheme, can be represented by a single activator. We will now demonstrate this tendency by an example, which is a generalization of an example used by Jones and Wulf in [Jon75].

An Example: Let memo be a type of object which carries memoranda. Suppose that in addition to the text itself, which can be retrieved by the operator read, every memo-object has a set of attributes

$$X = \{x_1, \dots, x_n\}$$

associated with it, where all x_i are boolean variables. We will say that a memo m "satisfies a certain attribute x_i " if $x_i(m) = \text{TRUE}$ (*). Suppose also that for every subject S there is a set $Y(S) = \{y_1, \dots, y_k\} \subseteq X$ of memo-attributes which represent his privileges with respect to memoranda, as follows: S should be allowed to read all memos, and only such, which satisfy all y_i in $Y(S)$.

An OC-representation of this policy is the following.

Let

$$\text{facade}(\text{memo}) = \{x_1, \dots, x_n\}$$

and let the primary read-activator be

$$\text{READ} = \langle \text{read}, [\text{memo}] \rangle$$

Suppose that a subject S is given only the following reduction of READ .

$$\text{READ}_1 = \langle \text{read}, [\text{memo};; y_1 \ \&, \dots, \& \ y_k] \rangle$$

Suppose also that the set of tickets $\{(m:\text{memo})\}$, one for each memo-object in the system, is stored on a file dir from which all subjects can copy tickets. It is obvious that the desired policy is satisfied under these conditions.

The salient feature of this implementation is that the various subjects have effectively different "power" with respect to memo-objects, due to the different read-activators in their domains. That is why they can safely share

(*)A possible interpretation of this is the following: There are n non-disjoint categories of memo objects. An object n belongs to the i -th category if $x_i = \text{TRUE}$.

the same set of tickets, contained in file dir, and still have different privileges. As we will see next, the situation is quite different under the AC case.

Under the AC-scheme, we assume that the operator read demands that the right "read" is in the operand-ticket. Since all subjects involved must have the right to invoke read, the difference between the subjects can only be in terms of the memo-tickets, each with the "read" right, which are available to them. Thus, the desired policy can be established as follows:

For a given memo-object m , let:

$$Z(m) = \{z_1, \dots, z_j\} \subseteq X$$

be the set of boolean attributes satisfied by m . Let target(m) be the set of subjects S such that for each of them

$$Y(S) \supset Z(m).$$

This is exactly the set of subjects which by our policy should be allowed to read m . Therefore, the ticket $(m; \text{memo}; \text{read})$ should be available to these subjects and to none other. In order to establish such a distribution of tickets, we suppose that every subject S in our system has a file, $\text{memos}(S)$, which is readable only by him. Whenever a new memo-object m is created, a non-copyable ticket $(m; \text{memo}; \text{read})$ should be stored in $\text{memos}(S)$ for every S in $\text{target}(m)$, and in nowhere else. This is essentially the solution given by Jones and Wolf to a similar problem [Jon75].

Let us now compare the control-material which is necessary for these two implementations of our policy:

As to the volume of the control material, suppose that there are NS subjects in a system, and M memo-objects. Let K be the average number of subjects which are allowed to read a memo-object. The AC implementation requires $K \cdot M$ memo-tickets to be stored in the system, while under the OC-representation only M tickets are required. (Also, the AC implementation needs NS tickets for the operator read, which is comparable to the NS activators needed under our scheme).

Even more important than the volume of the control material, is the complexity of its distribution. The AC-implementation requires a very specific distribution of the memo-tickets among the NS files memos(S). This distribution of tickets is itself a formidable task. Moreover, every file memos(S) must be well protected, and readable by the specified subject S only.

The situation under the OC implementation is much simpler: Once the NS different read-activators are correctly distributed among the various domain, we can store all the M tickets in one file, which is readable by everybody and does not have to be especially protected. This is obviously much less complex than under the AC-implementation.

4.3.2 Stability of the control material: Stability is a very desirable property of the control material, for a number of reasons:

- a) The maintenance of highly volatile control-material may take a lot of effort, particularly whenever it is involved with revocation.
- b) The probability for making mistakes in the distribution of control-objects increases with their volatility.

e) Stability of the control-material facilitates compile-time checking.

We will argue that the control-material tends to be more stable (less volatile) under our OC-scheme. This is due mainly to the following reasons:

a) Volatility clearly increases with the complexity and the volume of the control-material, which tends to be higher under the AC scheme.

b) There is a difference between the life time of tickets and activators. A ticket (b;r) cannot exist prior to the creation of object b, or after its destruction. An activator <o,[t]>, on the other hand, can live as long as the operator o and the type t exist; the meaning of this activator does not depend on the existence of any particular t-objects. Thus, an implementation of a policy under a AC scheme, which is based entirely on tickets, has an a priori temporary nature, and is therefore likely to be relatively volatile.

Moreover, the variations in control-material which do exist under the OC-scheme tend to concentrate in the transient part of domains, while their permanent part, which contains mostly activators, is relatively stable. This is important because it is mostly the permanent part of the domain of a subject which determines his authority (cf. section 3.6). We will now illustrate some of these observations in the context of our memo-example. We start by examining volatility under a fixed policy, the one presented in section 4.3.1, when the population of memoranda is changing.

Under our OC scheme, each subject S has an activator which determines the type of memo-objects which can be read by him. This gives S a general authority with respect to memoranda, which is independent of the particular memo-objects

in the system. Indeed, no change is necessary in the domain of S when new memo-objects are generated or when existing ones are deleted or changed. The authority makeup of S is stable under such changes. On the other hand, under the AC-implementation of this system, whenever a new memo-object is created its tickets must be given to all subjects who are allowed to read it. Even worse, when some of the attributes x_i of m are changed, the tickets of m may have to be revoked from those subjects which are not to be allowed to read it any longer. Thus the control material must be constantly changed to maintain the given policy.

Next, let us consider volatility under an incremental policy change. Using again the memo-system, suppose that in addition to read there is another operator, update, whose primary activator under the OC mechanism is:

<update,[memo], [text]>.

Suppose, also, that the set of memoranda that a subject is allowed to read may be different from the set he is allowed to update. For example, S may have the following two activators:

<read,[memo; x_1]>

<update,[memo;; x_1], [text]>

This means that S can read and update memos which satisfies x_1 .

Now, suppose that we wish to change this policy allowing S to update only memos which satisfy both x_1 and x_2 . All we have to do is to replace his update-activator with

<update,[memo;; $x_1 \& x_2$], [text]>

For the AC case, suppose that all the subjects have tickets which allow them to call both read and update, and that the operator update requires the right "update" from its argument just as read requires the right "read". Initially S must have access to the set of tickets

$$\{(m; \text{read}, \text{update}) \mid m\text{-satisfies-}x_1\}$$

for all memos which satisfy x_1 . To change the privileges of S as above we must replace all his memo-tickets with the two sets

$$C_1 = \{(m; \text{update}) \mid m\text{-satisfies-}x_1\text{-and-}x_2\}$$

$$C_2 = \{(m; \text{read}) \mid m\text{-satisfies-}x_1\}$$

Thus, the same policy change which requires the replacement of a single activator under a OC scheme requires the replacement of many tickets under AC scheme.

4.3.3 Efficiency of enforcement: Two factors effect the enforcement efficiency:

- a) The complexity of the computation which is necessary for the validation of a given operation.
- b) The degree to which validation can be performed at compile-time.

We already saw that the enforcement mechanism which is necessary to support the OC scheme is essentially identical to that of the AC scheme. In both cases, the operands must be checked against the operand-patterns of the given activator, which is the "template" in the case of Hydra. Of course, the complexity of such parameter checking depends on the complexity of the activation patterns. Our scheme allows for essentially arbitrarily complex patterns, but it does not require such complexity. If the OC scheme is used for the protection of operating systems, one would probably impose severe restrictions on the syntax

and semantics of activation-patterns and of G. Much more general patterns can be used in the context of information-systems, without a significant relative increase in the overhead due to protection.

As to the second factor which affects efficiency of enforcement, we claim that our scheme facilitates compile-time validation, due to the greater stability of its control-material. In particular, it appears that the compiler can do much of the necessary checking by analyzing the relatively static "permanent-part" of the domain of a subject. However, more study is necessary to substantiate this claim.

5: CONCLUSION

The operation-control (OC) scheme introduced in this paper is a natural generalization of the capability-based version of the access-control (AC) scheme developed for operating systems. This generalization is achieved by the introduction of the activators, which play an analogous role to that of the tickets under the AC scheme, and which do not require any new enforcement effort. The use of activators together with tickets has a profound effect on the authorization scheme: The representation of complex policies becomes easier and more natural. The control-material which is necessary for the representation of policies tends to be less voluminous, less complex and more stable. The stability of the control-material reduces the need for revocation, and facilitates compile-time enforcement. It is also believed that this stability and simplicity of the control material would facilitate the proof of policies. It should be pointed out, however, that the proposed scheme has yet to prove itself in the context of a real system. There is work in progress which attempts to base the protection of information systems on this scheme.

ACKNOWLEDGEMENTS

I wish to thank Joseph Stein from the Hebrew University of Jerusalem, David Levine and Matthew Morgenstern from Rutgers, Dorothy Denning from Purdue, and an anonymous reviewer; for reviewing this paper, and for their very useful comments.

REFERENCES

- [Coh75] Cohen E. and Jefferson D., "Protection in the Hydra operating system" Proc. of the 5th Symp. on Op. Sys. Principles, Nov. 1975.
- [Dah72] Dahl, O.J. and Hoare, C.A.R. "Hierarchical program structures," in Dahl, Dijkstra and Hoare, Structured Programming. New York: Academic Press, 1972.
- [Gra72] Graham, G.S. and Denning, P.J., "Protection-principle and practice", Proc. 1972 SJCC, AFIPS Press, 1972.
- [Har75] Harrison, M.H., et.al., "On protection in operating systems".
- [Jon75] Jones, A.K., Wulf, W.A., "Towards the design of secure systems", Software practice and experience 321-336 (1975).
- [Jon76] Jones, A.K. and Liskov, B., "An access control facility for programming languages", Carnegie Mellon U., Tech. Report, 1976.
- [Lam71] Lampson, B., "Protection," in Proc. 5th Princeton Symp. Information Science and Systems (Mar. 1971) pp. 437-443.
- [Lam76] Lampson, B. and Sturgis, S. "Reflections on an Operating-System Design", CACM May 1976.
- [Lin76] Linden, T.A., "Operating system structures to support security and reliable software", to be published in the surveys of the ACM.
- [Lis74] Liskov B. and Zilles S., "Programming with abstract data types." Sigplan notices, April 1974.
- [Min76] Minsky, N., "Intentional resolution of privacy protection in database systems", Comm. ACM, March 1976.
- [Min 76a] Minsky, N., "An Activator-based protection scheme", Rutgers Tech. Rep., July 1976.
- [Ros 76] Rosenblit, M. and Minsky, N. "On the decidability problem of the safety of protection systems", Rutgers U. Tech. Rep. February 1976.
- [Pop74] Popek, G. and Kline, C.S., "Verifiable secure operating system software" [AFIPS (1974 NCC), 145-151.], Proc. 1974 NCC, AFIPS Press, 1972, pp. 145-151.
- [Red74] D. Redell, "Naming and protection in extendible operating systems," Ph.D. dissertation, Univ. of Calif., Berkeley, 1974.
- [Sal75] Saltzer, J.H., and Schroeder, M.D., "The protection of information in computer systems", Proc. of the IEEE, Vol. 63, No. 9. Sept. 1975.
- [Wei69] Weissman, C., "Security controls in the ADEPT-50 time-sharing system," in 1969 FJCC, AFIPS Conf. Proc., vol. 35, 119:133.
- [Wul74] Wulf, W., HYDRA: The kernel of a multiprocessor operating system," Commun. ACM, vol. 17, 337:345, June 1974.
- [Wul75] Wulf, W. A., "ALPHARD: towards a language to support structured programs, CMU Tech. Rep. (April 74).

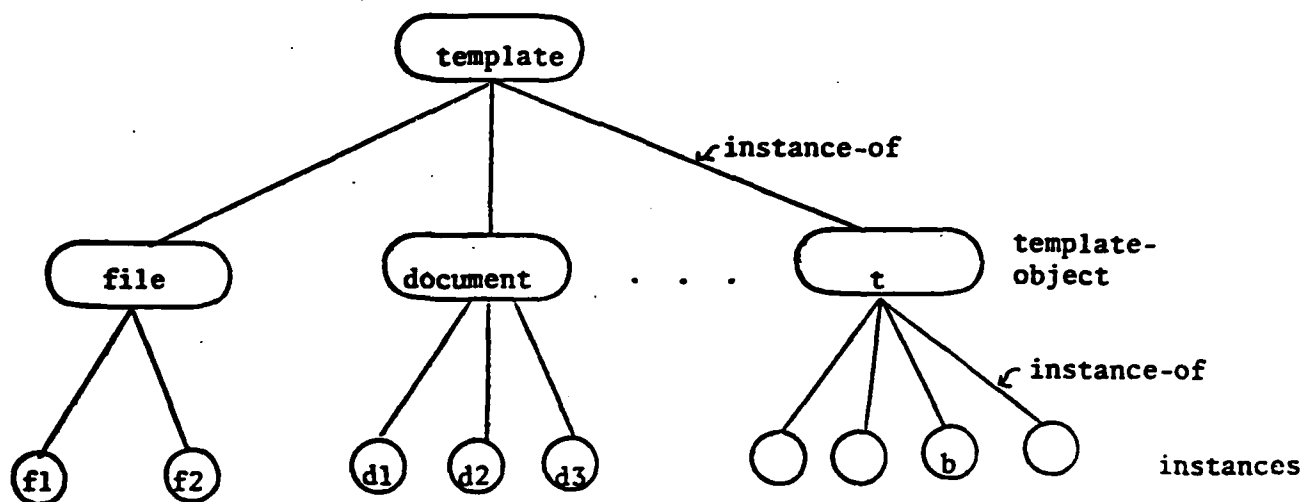


Figure 1: The Hydra approach to types

The object b is an instance of the object t (or a t-object). t in turn, is an instance of the distinguished object template, and thus it is a template-object (or type-object).

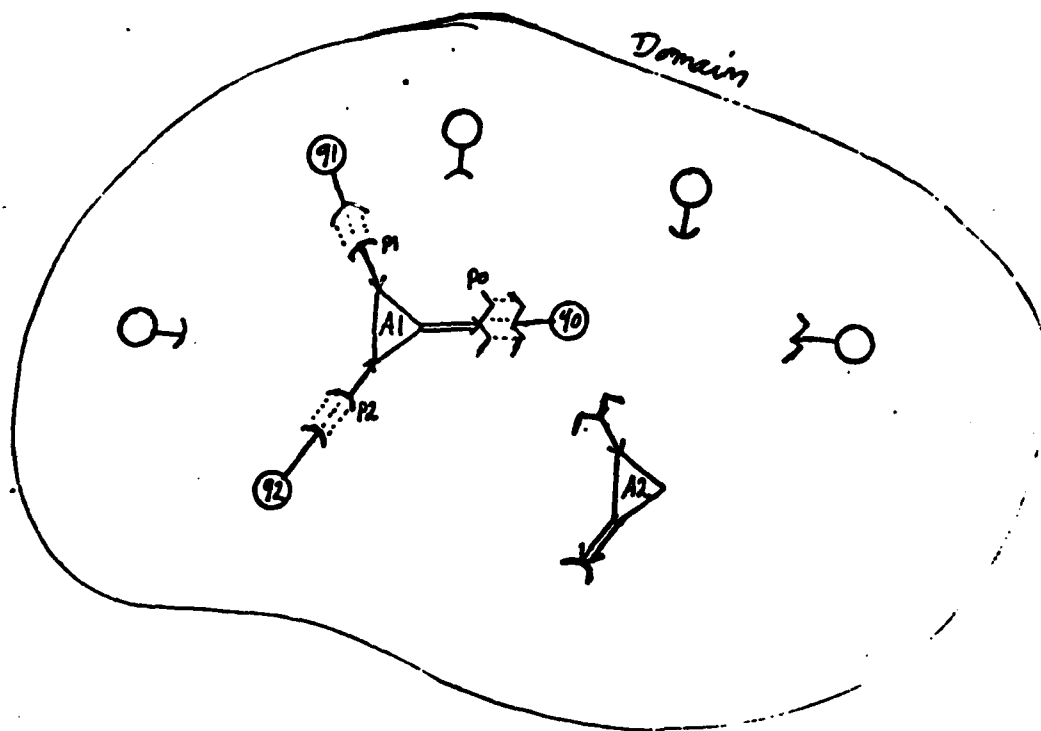


Figure 2

Activators in a domain are portrayed here as kind of enzymes in the living cell. The activator A1 is depicted as if it is attached to the objects q1, q2 which match its operand-patterns. The object q0, which matches the outcome pattern p0 of A1 is being generated. Note that q0 can be attached to the operand pattern of A2 and that the outcome of A2 would fit the pattern p2 of A1. The small circles represent various objects in the domain and the small patterns attached to them represent their type, facade and rights.

```

c1 = (f1:file)
c2 = (f2:file)
GET' = <getdoc,[f1],[text]> → [doc;u]
GET" = <getdoc,[f2],[text]> → [doc;E;slevel=1]
READ' = <read,[doc;;slevel<2]>
UPDATE' = <update,[doc;u;category="navy"],[text]>
ERASE = <erase,[doc;E]>

```

Figure 3: The permanent part of a domain

It contains two file tickets and five activators. GET' and GET" can operate on the file tickets, generating doc-tickets into the transient-part of the domain. The last three activators operate on these doc-tickets.

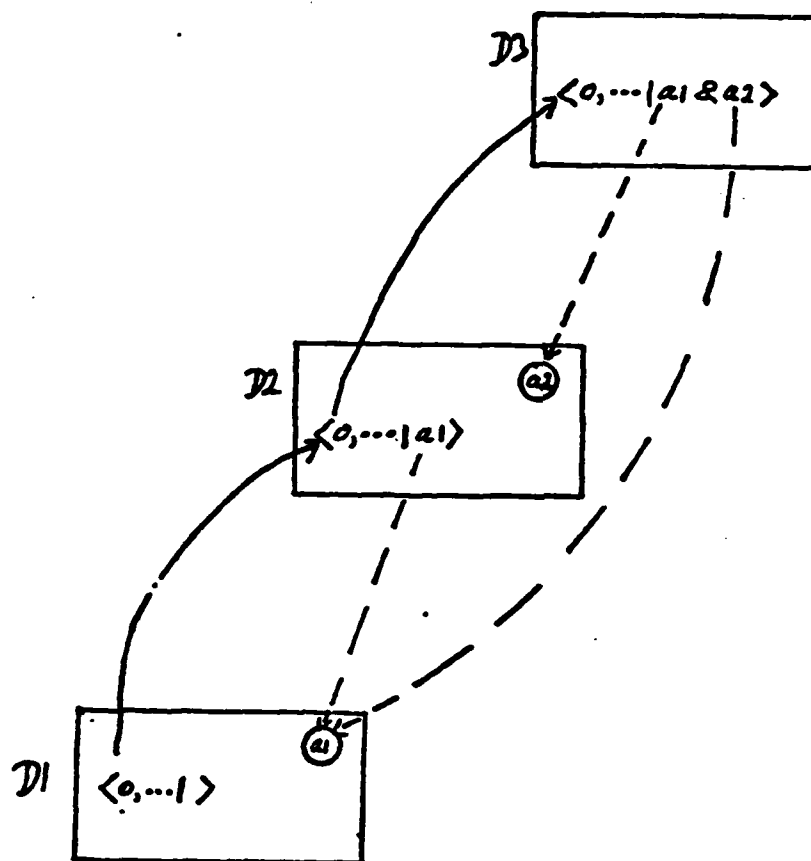


Figure 4: Revocation of Activators

The solid arrows represent sequence of derivation of activators. The dashed arrows represent dependency of the activator on the boolean variables a_1 , a_2 .

To be published in the Assoc. of the COMPSAC 77 Conference, November, 1977.

SOSAP-TR-34

May 1977

COOPERATIVE AUTHORIZATION IN COMPUTER SYSTEMS

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

ABSTRACT

The emerging technology of distributed computing, and the growing trend of computerization of large systems necessitates a technique for distributed and cooperative authorization and control. For example, a large financial transaction in a distributed electronic-fund-transfer (EFT) system may require the independent approval of several subjects. An authorization scheme which can support such a cooperation is described. It is based on the operation-control scheme recently developed by this author.

1: INTRODUCTION

Authorization in computer systems is a discipline under which an action on the system can be carried out by a user, or by one of the modules of the system, only if the actor is authorized to perform this action. Such a discipline facilitates the reliability of computer systems and is the basis for their protection. Cooperative authorization has to do with actions which have to be authorized by several subjects, independently. There are many real world situations which require such authorization. For example, the transfer of large sums of money from one account to another, or the release of sensitive information frequently requires the approval of several subjects. Another, unpleasant example for the need of cooperative authorization is the launching of nuclear weapons.

Cooperative authorization is essential in computer systems due to the ever increasing trend of computerization of large systems, such as information-systems, financial systems, medical systems, etc. The importance of such authorization is further enhanced due to the emerging technology of distributed computing, which facilitates applications such as the "electronic fund transfer". Unfortunately, in spite of the considerable interest in authorization and protection, cooperative authorization has not been studied systematically.

The purpose of this paper is to introduce the issue of cooperative authorization, and to suggest some techniques to support it. Our discussion will be in the context of the operation-control (OC) scheme recently developed by this author [1,2]. We start with an informal and partial overview of this scheme, its use for cooperative authorization is discussed in section 3.

2: THE OPERATION-CONTROL (OC) SCHEME

The OC scheme for authorization is a generalization of the capability-based version of the access-control scheme [3,4]. We start with a brief review of the OC scheme. Some extensions to it, which are necessary for cooperative authorization are introduced towards the end of this section.

A computer system changes in time due to instructions issued by subjects, where an instruction is a request to apply an operator to a sequence of objects-its operands. The authority of a subject is defined as the set of instructions which can be carried out by him. In our scheme this authority is determined by the content of the "domain" of a subject. A domain $D(S)$ is an object uniquely associated with the subject S which serves as the address-space of S . Namely, objects contained in $D(S)$ are directly addressable by S , and these are the only such objects. However, a domain may contain special objects called "tickets" which serve as pointers to "sharable objects" which do not reside in any domain, but can be shared by several subjects (*). In contrast to such sharable objects, the objects which physically reside in a domain are called "concrete objects". These include in particular tickets of sharable objects.

The main building blocks of authorization under the OC scheme are objects of type activator. An activator has the form(*)

 (*) Our tickets are equivalent to the "capabilities" of the access-control scheme [3,4]. Note that in addition to an object identifier, a ticket usually contains "rights" with respect to this object. However, for simplicity we will not use such rights in this paper.

(**) This is a simplified form of activators, see [1] for the complete form

$$\langle o, p_1, \dots, p_k \rangle$$

where o is an operator identifier, and p_i for $i=1, \dots, k$, are conditions on the operands of o , which are called the "operand patterns" of the activator. The existence of such activator in the domain $D(S)$ serves as a permission for S to apply operator o to a sequence of objects q_1, \dots, q_k in $D(S)$, which "match" the respective activation patterns (or satisfy the conditions) p_1, \dots, p_k . As a notational device, we will give a name, say " A ", to an activator by writing

$$A = \langle o, p_1, \dots, p_k \rangle$$

When an activator A is used to authorize the invocation of $o(q_1, \dots, q_k)$ we will say that the activator A is applied to the objects q_1, \dots, q_k , denoting such an application by $A(q_1, \dots, q_k)$.

An operand q which is matched with an operand-pattern, may be either a concrete object which stands for itself, such as an integer number, or it may be a ticket of a sharable object, which is the real operand. In both cases, we will refer to q as an operand, relying on the context to resolve the ambiguity.

An operand pattern p of an activator has the form

$$[I;;V]$$

where I is a condition on the type and identity of the operand, and V is a condition on its value. Instead of defining these components(*) of a pattern we will give a number of examples:

(*)The pair ";;" indicates a missing part in the more general form of the pattern introduced in [1].

Let doc be a type of objects which carry documents in a military information system. Suppose that in addition to the document itself a doc-object has two components: "slevel", which represents the "security level" of the document, and "category" which represents its category, such as "navy" or "army". Let read be an operator which displays the contents of a document. Consider the following activators for read, or, "read-activators":

$R = \langle \text{read}, [\text{doc}] \rangle$

$R1 = \langle \text{read}, [d:\text{doc}] \rangle$

$R11 = \langle \text{read}, [d:\text{doc} ;; \text{slevel} < 3] \rangle$

$R2 = \langle \text{read}, [\text{doc} ;; \text{slevel} < 2] \rangle$

$R21 = \langle \text{read}, [\text{doc} ;; \text{slevel} < 2 \ \& \ \text{category} = \text{"navy"}] \rangle$

The activator R can be applied to any document; $R1$ can be applied only to the specific document d ; $R2$ can be applied to any document whose $\text{slevel} < 2$; $R21$ can be applied to any "navy" document whose $\text{slevel} < 2$; finally, $R11$ can be applied only to document d , provided that its slevel is smaller than 3.

Note that all these read-activators are different from each other. In a sense, they have different power. In order to compare the power of different o-activators for a given operator o , we now introduce a number of concepts.

Let A be an activator of order k (with k operand-patterns). We define range(A) to be the set of all possible k -tuples (q_1, \dots, q_k) of objects, which can be matched with the corresponding activation-patterns of A .

Let A and A' be two o-activators for a given operator o . We will say that A' is weaker than A (or, equivalently, A is stronger than A') iff

$$\text{range}(A') \subseteq \text{range}(A).$$

such an A' will also be called a reduction of A .

Note that the relation "weaker" defines a partial order on activator. For instance, in the example above, $R1$ and $R2$ are weaker than R , but not of each other. Also, $R11$ is weaker than $R1$, and $R21$ is weaker than $R2$.

Here is a summary of the main properties of activators, which are discussed in greater detail in [1].

- a) When a new operator o is created, an o -activator is created with it. It is called the primary o -activator. We will see below that all other o -activators are derived from the primary one.
- b) The only legal change of an activator is its reduction.
- c) An activator can be transported from one place in the system to another by means of two "transport-operators" k-copy and k-move(*). When one of these operators is applied to an existing activator A , it generates a new activator A' which is either identical to or weaker than A . A' is called a derivative of A . (The main difference between these two operators is that k-copy does not affect the original activator A , while k-move erases it. Thus, in effect, k-move moves an activator A from one place to another).

The following corollaries follow directly from the above:

- 1) Every activator is stronger than all its derivatives
- 2) The primary o -activator is the strongest o -activator.
- d) Every activator has two boolean attribute "copy" and "move" which control its transportability. An activator A can be copied only if its "copy" attributes is true and it can be moved only if its "move" attribute is true. If any one of these attributes of an activator A is false, it will be false in all its derivatives.

*)The letter k indicates that these two operators belong to the "kernel" of the authorization scheme [1].

e) An activator A may have a usage(**) attribute $u(A)$. If this attribute exists it sets a limit on the number of applications of A . (An activator whose usage component is one will be called a non-reusable activator). In conclusion we would like to reiterate that in a system based on the OC scheme no action can be performed without an appropriate activator. Moreover, it should be emphasized that the generation and transport of activators is tightly controlled so that the ownership of an activator A by a subject S can serve as an uncontestable proof that S is authorized to perform any action enabled by A .

So much for the oc scheme, as it is defined in [1]. We will now introduce two minor extensions to it.

Extension 1: The application of an activator A is a complex indivisible operation which consists of the binding of operands to the operand patterns of A , and then the invocation of the operator q addressed by A . We will now allow for the separation of these two parts of the application of activators, as follows:

first, we introduce the operator

$$\text{bind}(A, i, q)$$

which binds the object q to the i -th pattern p_i of activator A , provided that q matches this pattern. This operation has a permanent effect on A and on q , in the following sense: Once an object q is bound to an activation pattern p_i by

(**)The "usage" facility has been introduced in [1] but in a different form.

the operator bind, it cannot be removed from it. This means that q is not free for use, and that the pattern p_1 of A cannot be bound to any other object. This is in contrast to the indivisible application of an activator, which leaves the patterns free to be bound to other objects on successive applications of the activator. Accordingly, binding by means of this operator will be called "permanent binding".

Now consider the activator

$$A = \langle o, \underline{p_1}, \dots, \underline{p_k}, \dots, p_n \rangle$$

whose patterns p_1, \dots, p_k are permanently bound. (To distinguish between bound and unbound patterns we underline the formers). Such an activator can be applied to objects $q(k+1), \dots, q_n$ which will be bound (not permanently) to the still unbound patterns $p(k+1), \dots, p_n$. In the special case of $k=n$, namely when all the patterns are permanently bound, the application of the activator is simply the invocation of the operator q on the already bound operands.

For the purpose of permanent binding we will allow for limited sharing of activators as follows: An activator A which resides in a domain $D(S)$ may be shared by several subjects who can bind operands to it. However, only the subject S who has A in his domain can actually apply it.

Extension 2: Given an activator

$$A = \langle o, p_1, \dots, p_k \rangle$$

We will allow for the following derivative of A :

$$A' = \langle o, p_1, \dots, p_k, p(k+1) \dots, p_n \rangle.$$

A' is identical to A except that it has $n-k$ new operand pattern. A' will be considered weaker than A because it imposes stronger requirements on the invocation of the operator o , namely, the availability of objects $q(k+1), \dots, q_n$ which match the new patterns.

3: COOPERATIVE AUTHORIZATION

Consider a subject S_1 who has the o -activator

$$A = \langle o, p_1, \dots, p_k \rangle$$

in his domain but who does not have a complete sequence of objects q_1, \dots, q_k which fit the respective patterns p_1, \dots, p_k of A . Suppose, however, that such a sequence exists in the union of the domains of the subjects S_1, \dots, S_n . It is clear that S_1 by himself cannot apply A , but A can be applied by means of cooperation between all the subjects S_1, \dots, S_n . One can think of two basic modes of such a cooperation:

First, the subjects S_2, \dots, S_n may give S_1 the objects he needs for the application of A . Unfortunately, this simple mode of cooperation has the following serious drawback: Let S_2 be a subject which gives S_1 the object q_2 which is necessary for the application of A . The problem is that S_1 may have several activators which can be applied to q_2 , so that S_2 has no way of knowing how would his object q_2 be actually used by S_1 . Such a cooperation may therefore be dangerous for S_2 .

The second mode of cooperation which does not suffer from the above problem, is the following: S1, who wishes to apply the o-activator A, first creates the weakest reduction A' of A which still suits his purpose. A' is now given to each of the subjects S2,...,Sn who are asked to make their contribution by binding(*) one or more operands to the operand-patterns of A'. This binding is permanent, which means that an object q bound to a pattern p of A cannot be misused by S1 since it cannot be freed from this binding and cannot be used for any other purpose. Moreover, each of these subjects can examine all the patterns of A' and the operands already bound to them, if any, so that he has a way to know the nature of the action in which he cooperates. Note that the degree of knowledge that each of the cooperating subjects has about this action depends on the degree of specificity of the operand-patterns of A'. That is why it is important for S1 to create the weakest possible, the most specific, activator A'. If A' is not weak enough some of the subjects S2,...,Sn may refuse to cooperate in its application. This mode of cooperation will now be illustrated by means of two examples.

3.1: The Signature Example

Let acc be a type of objects which represent accounts in a computerized financial system. Suppose that the only operator which can change the amount of money in an account is the operator move whose primary activator is (**)

MOVE = <move,amount:[int], from:[acc],to:[acc]>.

 (*)Note that this binding can be done in parallel, if the subjects S2,...,Sn "share" the activator A'.

 (**)The phrase "amount:[int]", gives the label "Amount" to the associated pattern. Such a label, which is used only for discussion, is optional.

The operation $\text{MOVE}(k, a_1, a_2)$ moves k dollars from account a_1 into a_2 , (thus preserving the total amount of money in the system(***)).

The activator MOVE is very powerful, since it can be used to move money between any two accounts. We assume, however, that there is just one copy of MOVE in the system, contained in the domain of the account-generator. It is also assumed that together with an account a_1 , the account-generator creates the following derivative of MOVE :

$$\text{MOVE-}a_1 = \langle \text{move}, \dots, \text{from}:[a_1:\text{acc}], \dots \rangle$$

The only difference between MOVE and $\text{MOVE-}a_1$ is that the "from" pattern in the latter can be matched only to the specific account a_1 . Thus, $\text{MOVE-}a_1$, which is weaker than MOVE , can be used to move money from a_1 to an arbitrary account.

Next, let us introduce the type signature. A signature-object, or simply signature, is an object which contains the identifier of the subject which generated it. A signature can be generated only by means of the operation

$$\text{sign}(A, i)$$

which creates a new signature and binds it to the i -th pattern of activator A , thus, in effect, "signing" the activator A . Of course, this can be done only if the i -th pattern of A matches the signature of the subject who invokes the operation above. Note that a signature thus defined has no free existence in

 (***) To represent flow of money into and out of the system one can introduce special "source" and "sink" accounts (see [2]). However, we will not be concerned with this issue here.

the system because its generator always binds it permanently to some activator.

Now, suppose that the subject S, who generated a certain account a_1 , decides that every removal of money from a_1 must be approved by two subjects S1 and S2. This policy can be enforced as follows. Suppose that S replaces his MOVE- a_1 activator with the following reduction of it.

$$\text{MOVE-}a_1' = \langle \text{move}, [\text{int}], [a_1:\text{acc}], [\text{acc}], \\ [\text{signature};;S1], [\text{signature};;S2] \rangle$$

This activator contains two new patterns which match only signatures of S1 and S2, respectively. Thus, every removal of money from a_1 , which requires the application of MOVE- a_1' or of one of its derivatives, must be endorsed explicitly by the signatures of S1 and S2. One can imagine the following sequence of events which lead to the application of MOVE- a_1' : A subject S' who wishes to move k dollars from a_1 to a_2 , generates the following non-reusable reduction of MOVE- a_1'

$$\text{MOVE-}a_1'' = \langle \text{move}, [\text{int};;\text{val}=k], [a_1:\text{acc}], [a_2:\text{acc}], \\ [\text{signature};;S1], [\text{signature};;S2] \rangle$$

which can be used only to move k dollars from a_1 into a specific account a_2 . This activator is now sent to S1 and S2 for approval. Each of these subjects can examine this activator, and if he approves of the implied transaction he would endorse it by "signing" the appropriate pattern. Once the activator is signed by both subjects it can be applied to the operands k, a_1 , and a_2 . Note that S1 and S2 can sign the activator in parallel, if they share it.

A number of variations of this approval process are possible. We will mention two of them. First, the subject S1 and S2 may be ready to approve a more general transaction than the move of k dollars from a_1 to a_2 . For example, they may be ready to sign the activator

$\langle \text{move}, [\text{int};; \text{val}=\text{k}], [\text{a1}], [\text{acc}], \dots \rangle$

which can be used to move k dollars from a1 to any account. The point is that S1 and S2 can examine the activator before they sign it, so that each of them knows what kind of action he approves.

A second variation of our example is the following. Suppose that every signature-object created by a subject S is a pair (S,l) where S is the identifier of the subject, as above, and l is its "signature-level". the signature-level is a number associated with a subject which reflects, in some sense, his importance as an officer in the corporation. Now, one may require that a transaction be approved by the signature of one or more subjects with a given signature-level, which is a very common requirement in banking. For example, the activator.

$\langle \text{move}, \dots, [\text{signature};; \text{level} \geq 2] \rangle$

cannot be applied without a signature whose level is at least 2.

3.2: The medical-prescription example(*)

(*)This example has been chosen for its intuitive appeal. Unfortunately the example implies a somewhat unrealistic degree of computerization of health management. However, it is easy to find more realistic examples which have similar authorization structure.

The act of selling a drug to a patient is involved with a fairly complex, cooperative authorization procedure. Three subjects actively cooperate in this act. First, there is a doctor D who gives a prescription for a specific drug d to patient P . This prescription serves as an authorization for P to buy the drug d . Moreover, the prescription may serve as an authorization to charge the drug-prescription-program of P for the prescribed drug. Next, the patient can bring the prescription to a pharmacy of his choice. The pharmacist, who has access to drugs but no rights to sell them to specific people, is thus authorized to sell to patient P a drug which fits the doctor's prescription. (Note that the doctor may specify the chemical name of a drug, leaving it up to the pharmacist to choose a brand name).

For the sake of this example, imagine that there is a robot in every computerized pharmacy, which actually gives drugs to customers. This robot can be activated only by means of the operator sell whose primary activator is

$$\text{SELL} = \langle \text{sell}, [\text{drug}], [\text{patient}], [\text{acc}] \rangle$$

As we will see below, this activator is equivalent to a blank prescription form, and we assume that only doctors have copies of it.

Now, a doctor creates the equivalent of a prescription by generating the following reduction of SELL

$$\text{SELL}' = \langle \text{sell}, [\text{drug};; \text{cname}], [D], [\text{acc}] \rangle$$

The first operand-pattern of SELL' matches only drugs whose chemical name is cname, while its second pattern is bound to a ticket p of a specific patient P .

This "prescription" is now given to P who binds its third pattern to the ticket of some account ac. This may be a general purpose account, such as BankAmericard, or, it may be the patient's prescription-program account. The resulting activator is now given to some pharmacy. There, finally, SELL' is applied to a drug whose chemical name is cname, giving P his drug and charging the account ac for it.

Note that the real-world policies with respect to selling of medicines impose some additional restrictions. First the "prescription" SELL' given by a doctor to his patient, must be uncopyable, although it should be transferable. Moreover, it should be unreusable, or at least it should have a finite usage, which is equivalent to the number of refills allowed by the doctor.

4: CONCLUSION

This paper discussed the issue of cooperative authorization purely in the context of the OC-scheme. It would be more difficult, if at all possible, to base such authorization on the conventional access-control scheme, as it appears in Hydra [4] for example. However, some recent generalizations of the capability-based scheme [5,6], which features more complex tickets structure appear to facilitate at least a certain degree of cooperative authorization, although the authors of these schemes did not discuss such authorization explicitly.

Acknowledgments: I would like to thank M. Morgenstein from Rutgers and J. Stein from Harvard for thier constructive criticism of the ideas expressed in this paper.

REFERENCES

1. Minsky, N., "An operation-control scheme for authorization in computer systems", Rutgers Tech. Rep., April 1977.
2. Minsky, N., "An Activator-Based Protection Scheme", Rutgers Tech. Rep., July 1976.
3. Saltzer, J.H. and Schroeder, M.D., "The Protection of Information in Computer Systems," Proc. of the IEEE, Vol. 63, No. 9, Sept. 1975.
4. Wulf, W., et. al. "HYDRA:EEthe kernel of a multiprocessor operating system", Comm. ACM 17,6, (June 1974), 337-345.
5. Saal, H.J. and Gat, I., "A hardware architecture for controlling information flow", IBM Res. Report, RC 6414 (Feb. 1977).
6. Ekanadham, K., and A.J. Bernstein, "Access control using contexts", Stony Brook Tech. Report. 1976.

SOSAP-TR-35-A

July, 1977

AUDITING OF COMPUTERIZED FINANCIAL SYSTEMS

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

Abstract

The internal control of a financial entity is based on complex, sometimes subtle, system of authorization and capabilities which we call the "authority structure" of the entity. The computerization of a financial entity carries with it a danger of the breakdown of its authority structure, which might endanger the internal control of the entity, and its auditability. It is our thesis, however, that with a proper use of authorization and protection techniques developed for computer systems, it is possible to impose a desired authority-structure on a computerized financial entity. The feasibility of such computer based authorization and its ramification for auditing are the subjects of this paper.

1: Introduction

The auditor's analysis of a system depends to a large extent on his knowledge, or beliefs, about its "authority structure". By this we mean the authority and capabilities of the various actors, or subjects, which play active roles with respect to the system. For example, the reliance of the auditor on any documentary evidence, such as the audit trail, depends on his belief that the subjects who might have any interest in forging documents, did not have any access to them. Indeed, the authority structure of a system is the linchpin of its "internal control" and it provides the basis for concepts such as segregation of duties, and for tools such as the audit trail.

The nature of the authority structure of non-computerized financial systems is well understood. The authority structure of a given system can be determined by means of questionnaires, interviews, observation of personnel and review of documents pertaining to corporate structure, such as organization charts, etc. There are well established means for the enforcement of adherence to authorization; they include physical limitations, legal codes, social pressures, and the psychological makeup of people. There is even some knowledge about the ways in which humans are likely to err or otherwise commit improprieties, as well as the circumstances under which these events can take place. Actual compliance to authorization can be determined to a great extent by observing people and inspecting records using the audit trail.

The issue which will concern us in this paper is that the computerization of a system brings with it the danger of a breakdown of its authority-structure, as well as the opportunity for tightening it up, making the system safer and more auditable. In the next section we will explain briefly why computerization endangers the authority structure of systems. Our approach for avoiding these dangers, and for capitalizing on the opportunities of computerization are discussed in section 3.

2. The breakdown of the authority structure of conventional computerized systems

The prime effect of computerization is that much of the activity of the system is performed by program modules (procedures, subroutines, etc.) rather than by people. This might seem only to improve the situation because one can rely on a program to do exactly what it is programmed to do. Unfortunately, it is virtually impossible to know exactly what every one of the hundreds of program-modules of a system will do under all conceivable circumstances, even under the (unrealistic) assumption that these modules themselves are not being changed during the time period studied by the auditor. Thus, the auditor must treat the various modules of the system as black boxes whose internal structure, their code, is partially or completely unknown.

Now, the problem is that if we do not know exactly how a given module is programmed, there is nothing we can say about what it might do. Potentially, every program-module has unlimited power with respect

to the data-base maintained by the system. For example, a module that is designated to update the personnel file, might actually be programmed, inadvertently or maliciously, to reach the accounts receivable file, changing it in some unpredictable fashion. As another example, the auditor may be told that certain types of transactions leave an audit trail, stored on a given file F. He might even be able to validate this claim by a careful study of the module which carries out the transaction. But how does he know that the file F has not been changed by some other module of the system? In a manual system, the auditor is usually able at least to identify a small set of people who are physically able to get to a filing cabinet, but in the computerized case every module is able to update F. Thus, even the credibility of the audit trail, one of the main tools of auditors, is destroyed.

↪ A similar problem exists in the case of a user, which is a person who interacts with the system: He is not restricted by many of the physical restrictions which may limit his power in a manual system. Unless something is done about it, the user would have virtually unlimited power with respect to the system at hand. Moreover the probability of sinister use of this power is enhanced due to the following two reasons: First, a user who interacts with a computer is not inhibited by the normal social pressures which exist in interaction between people (unless he suspects that his interaction with the computer is recorded). Secondly, the incentives for fraud are stronger in a computerized system due to the high degree of centralization of resources. The procedures that would inhibit a person from stealing \$500 are likely to be less effective when an opportunity to steal 50,000 presents itself.

These remarks lead to the conclusion that one must be able to impose a priori limitations on the power of the various subjects (actors) of the system, whether they are program modules or people which interact with them. For example, a program module P which is supposed to update the personnel file should be invested with sufficient authority to do so. However, P should not be able to access the accounts receivable file, or any other part of the system which is irrelevant to its function, regardless of the actual code in P. The discipline which allows one to impose such restrictions on the subjects of computerized systems, is called "authorization mechanism", to be discussed next.

3: Authorization in computerized systems, and its use for auditing

Conceptually, an authorization mechanism consists of two parts: First, there is the authorization scheme which is a formal language which allows us to specify the authority of each and every subject in the system, whether it is a program module or a person. Secondly, we need an enforcement-mechanism to protect the system against unauthorized operations.

The following example illustrates a way in which authorization could improve the auditability of systems. We will be using the "operation control" authorization mechanism [1,2] recently developed by the author. This mechanism and its possible applications will be discussed in some detail in the full paper.

An example: One of the most important types of entities in any

financial system are accounts. An account might represent various items like assets, liabilities, revenues (all in dollar units) and it should be accessible only by a specified set of subjects. Under a computerized information system. The accounts would be represented simply by records on a file, and in the absence of authorization, every subject would be capable of changing the content of every account, in an arbitrary way. This state of affairs must raise some doubts in the auditors mind as to the authenticity of the accounts which he reviews, particularly, because erroneous or malicious changes of accounts are not likely to leave any audit trail. However, using the OC mechanism it is possible to establish the following type of discipline, (see [1] section 2.8).

- 1) The only module in the system which can update accounts is the procedure move(a1,a2,k) which moves k dollars from account a1 to a2, leaving the total amount of money in the system unchanged. To represent the flow of money into the system we will have a special procedure gen-account(k) which generates a new account with k dollars in it. To represent outflow of money, we will have a special type of accounts called sinks which have the property that money can flow into them but not back. It is assumed that the procedures move and gen-account are correctly programmed, and that they are designed to leave an audit trail of their actions.
- 2) The use of the procedures move and gen-account by the various subjects in the system can be regulated in such a way that it is possible to specify who can move money between which

accounts and who can generate new accounts.

From the fact that the procedures move and gen-account are the only subjects which can effect accounts directly. We can derive the following conclusion: The total inflow of money into the system minus the total outflow, is equal to the total amount of money in the system (the sum of all non-sink accounts). This "law of conservation of money" can be of enormous help for auditing. One knows that the money in the system cannot appear from nowhere, nor can it disappear into thin air, and that every generation or movement of money is properly documented in the audit trail. Moreover, the auditor should be able to tell who can move money from a given account, and he should be able to predict possible routes of money flow. For example, if there is a suspiciously large sum of money in a given account, the auditor knows at least that this money has been moved from some other account in the system, and is not due to some wild update of the record which represents the account. Moreover, the auditor should be able to identify the set of subjects who might be responsible for such a move, even without looking on the audit trail. [end-of-example]

REFERENCES

1. Minsky, N., "An operation-control scheme for authorization in comopter systems",o be published in the "Int. Journal of Comp. and Inf. sci."
2. Minsky, N., "An Activator-Based Protection Scheme", Rutgers Tech. Rep., July 1976.

THE PRINCIPLE OF ATTENUATION OF PRIVILEGES
AND ITS RAMIFICATIONS

N. Minsky

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research
Projects Agency of the Department of Defense under Grant #DAHC15-73-G6
to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the
author and should not be interpreted as necessarily representing the
official policies, either expressed or implied, of the Advanced
Research Projects Agency or the U. S. Government.

Abstract

This paper argues that an authorization scheme should, and can, be based on the "principle of attenuation of privileges". It is shown that the well known access-control scheme is incompatible with this principle, due to its failure to distinguish between "privileges" and "abilities". The operation-control scheme, which is based on the principle of attenuation, is described. The issue of type-extension is discussed from the point of view of Hydra and of the proposed operation-control scheme. The implications of the principle of attenuation to the formal protection models introduced by Harrison, Lipton and others are briefly discussed.

Introduction

Authorization in computer system is a discipline under which it is possible to impose restrictions on the kind of action which can be carried out by the various actors, usually called "subjects", of a system. One can distinguish between the "statics" and the "dynamics" of an authorization scheme. By the terms statics we mean the method used for the representation of the authority of the various subjects, as well as the technique for the enforcement of such authority. What we call the "dynamics" of the authorization scheme is the techniques used for the manipulation of the "authority-state" itself.

The traditional approach to the design of authorization schemes seems to start with its statics and then design a suitable dynamics for it. In particular, the well known access-control AC scheme started from the observation that one needs in operating systems is to specify for each subject which operation, if any, he can apply to each object in the system. This led to Lampson's access-matrix model [Lam 69], and to the theoretically equivalent "capability-based" model [Fab 68]. Studies of the dynamics of authorization followed these developments, and have been mostly based on the access-control approach previously formulated. These include the work of Graham and Denning [Gra 72], Jones [Jon 73], and recently the more formal studies of dynamics by Harrison et.al. [Har 76], and by Lipton et.al. [Lip 77]. These studies revealed some serious difficulties such as the revocation problem, the undecidability of the safety problems, and, what will concern us in this paper, the apparent need to perform amplification of privileges. This situation suggests to this author that some of the difficulties with the dynamics of authorization is due to the choice of statics on which it is based.

The approach of this paper is to start with some basic principles of the dynamics of authorization, which would lead us to certain conclusions about its statics. More specifically we will take the principle of "attenuation of privileges" as our basic premise. Informally speaking, this principle means that privileges should not be allowed to grow when they are moved from one place in the system to another. This principle has been suggested by Denning [Den 77] and has been recently the subject of some controversy. We believe that this principle is vital for our ability to prove correctness of policies, and we will show that the access-control scheme is inherently incompatible with it. The wish to satisfy the principle of attenuation will lead us to a new scheme called operation-control (OC), which has been previously introduced [Min 77] for a number of different reasons. The operation-control scheme, which is squarely based on the principle of attenuation, can be viewed as an extension of the capability-based version of the AC (access-control) scheme.

In the next section the capability-based version of the AC scheme is described. The principle of attenuation is formally defined, in the context of this scheme, and the inability to satisfy it is demonstrated. The underlying reasons for the incompatibility of the access-control scheme with the principle of attenuation is discussed in section 3. In section 4, some aspects of the OC (operation-control) scheme are introduced, just enough to show its compatibility with the principle of attenuation. A comparison between the OC scheme and the scheme used in Hydra system is made in section 5, and the implementation of "type extension" under the two scheme is discussed in section 6.

2: The Access-Control (AC) Scheme

The access-control approach to authorization is well documented in the literature. In particular, the reader is referred to the work of Lampson [Lam 69, Lam 71], Graham and Denning [GRA 72], and to recent review articles by Saltzer [Sal 75], and Linden [Lin 76]. Here we will outline the main features of a class of AC schemes called capability-based [Fab 68, Wul 74], using a somewhat non-standard terminology which is more suitable for the rest of the paper. We start with the statics of the capability based scheme. (The scheme to be described above differs in an essential way from the scheme used in Hydra. Hydra is discussed in section 5).

The objects to be protected by the system are classified into types. For the moment we will assume that every object belongs to a unique type. An object of type T will be called a T-object. For every type T there is a fixed set of operators (procedures)

$$op(T) = \{P_i\}$$

called T-operators. It is assumed that T-operators are the only subjects in the system which can directly manipulate and observe T-objects. For all other subjects the only way to manipulate or observe T-objects, is indirectly by applying to them T-operators. (We will see later how this rule can be enforced by the protection system itself, for all but a fixed set of primitive types.)

Also, for every type T there is a fixed set of symbols

$$rt(T) = \{r_i\}$$

called T-rights, or simply rights. Objects of type T (T-objects) are addressed by special kind of objects called tickets (*), which have the form

(*) We are using the term "ticket" for what is more commonly called "capability". The reason for this deviation from the, more or less, standard terminology will be clarified later.

(b;R)

Where b is the identifier of a T-object and R is a subset of $rt(T)$. There may be several tickets in the system with the same component b , they will be called b-tickets. The right-symbols contained in a b-ticket t serve to determine which T-operators can be applied (*) to b , when the ticket t is used to address it. It is in this sense that a ticket represents privileges with respect to the object addressed by it. For the rest of this section we assume the following one to one correspondence between T-rights and the T-operators which they authorize:

The T-operator P_i can be applied to a ticket $(b;R)$ of a T-object b , only if R contains r_i .

In such a case r_i may be called "the right for P_i ". Although the correspondence between rights and operators may be more complex than that, it is always monotone, in the following sense:

If an operator (P_i) can be applied to a ticket $t=(b;R)$, then it can be applied to any ticket $t'=(b;R')$ such that R' includes R .

This monotone property suggests the following relation which defines a partial order between tickets.

Definition: A ticket $t=(b;R)$ is weaker than $t'=(b;R')$, if R' includes R .

(*) Since objects are always addressed to by their tickets we will frequently use the phrase "application of an operator to a ticket", to mean the application of the operator to the object which is addressed by the ticket.

(T) We frequently use the terms "operator" "object" and "right", for "T-operator", "T-object" and "T-right" when the identity of the type T can be understood from the context, or is not important.

Clearly, a weaker ticket carries less privileges.

Now, every subject (actor) in the system is associated with a special kind of object which we call domain (in Hydra [Wul 74] it is called the LNS, for "Local Address Space", of the subject). The domain D of a subject S contains tickets of various objects in the system, and it is assumed that a subject can operate only on tickets in his domain. In this way the domain of a subject determines his authority.

The distribution of tickets through the system is called the authority state of the system. The dynamics of the authority state is discussed below.

2.1: On the Dynamics of the AC Scheme

Although there is no general agreement as to the ways in which the authority state of the system is to be changed, the dynamics of most AC scheme is governed by the following rules. (Even if these rules are not explicitly stated, or stated in a different form.)

Rule 1: An existing ticket cannot be modified.

Rule 2: When a T-object t is created, a ticket $(t;R)$ is created with it, with all its possible rights. (Namely $R = rt(t)$). We will call it the primary t -ticket.

Rule 3: There is an operator, duplicate, which when applied to a t -ticket t , creates another t -ticket t' in some other place in the system. t' will be called a direct derivative of t . (We will use the term "derivative" of a ticket t for a direct or indirect derivative of t .)

To these, practically standard rules, we now add another rule which turns out to be controversial.

Rule 4: The direct derivative of a ticket t cannot be stronger than t .

This means, in particular, that the primary b -ticket is the strongest b -ticket, for every object b .

This last rule is essentially what Denning [Den 76] called the principle of attenuation of privileges. Although, as we will see later, this is an enormously useful principle, it is violated by some AC schemes, notably by Hydra [Wul 74, Cah 75]. One of the main features of Hydra is an operator amplify which when applied to a ticket t , creates a ticket t' which is stronger than t ^(*). Even Denning who was the first to suggest explicitly the principle of attenuation, qualifies himself by requiring it only "under normal circumstances" [Den 76, p. 372] (see also the debate between Denning and Levine in the June 77 issue of Computing Surveys [Den 77, Lev 77]).

Indeed, it turns out that the principle of attenuation is incompatible with the AC scheme. To see this consider the following example.

Example 1: Let r_1, r_2 be T-rights, for a given type T , and let P_1, P_2 be T-operators such that P_i can be applied only to a ticket with the right r_i , for $i=1,2$. Now, consider the following policy with respect to two subjects S_1, S_2 and a T-object b .

- 1) S_1 is allowed to apply P_1 but not P_2 to b .

(*) Actually, Hydra allows only a restricted use of the operator amplify. We will discuss Hydra specifically in a later section.

2) With the approval of S1, S2 would be allowed to apply P2, but not P1, to b.

We will now see that without amplification these simple requirements cannot be satisfied under a AC scheme, as it has been described above.

To satisfy requirement 1, S1 must be given a ticket $t=(b;r1)$. Note that t cannot contain $r2$ because this would allow S1 to apply P2 to b. Now, how does S1 approve the application of P2 to b by S2? The most he can do is to give to S2 a derivative t' of the ticket t which he has himself. However, due to the principle of attenuation t' cannot contain $r2$, which is necessary for the application of P2. Moreover, S2 himself cannot have a ticket for b with $r2$ in it because this would allow him to apply P2 to b, without the approval of S1. Thus, our policy cannot be realized.

This is the kind of problems which prompted Jones [Jun 73] to introduce the amplification operator violating the principle of attenuation. In this case amplification would allow S2 to add the right $r2$ to a ticket for b given to it by S1. Our approach to this dilemma is the opposite: taking the attenuation of privileges as a fundamental principle of an authorization scheme, we conclude from this example that the AC scheme itself is unsatisfactory and should be replaced by a scheme which is compatible with this principle. But first we must gain a deeper understanding of the source of the difficulty demonstrated above.

3: Privileges versus abilities

Authority structures such as in example 1 are very common in the real

(*) In the real world, ability also depends on such things as skill and stamina of the subject, which we have no intention to model.

world, and it would be instructive to see how they are handled. Let us consider

^{one}
 ✓ such real-life example.

Example 2: When buying a car one automatically gets the right to drive and to sell it. These rights can be formally represented by a ticket-like structure (c;drive,sell) which stands for the ownership document for the car c. Now, consider a subject S1 who owns a car c, but who does not have a driving license. This person cannot exercise his right to drive his own car. However, S1 can hire a driver, who does have a driving license, authorizing him to drive the car c by giving him the ticket (c;drive). No process which even remotely resembles amplification is taking place in this real-life situation. The driver S2 can drive the car owned by S1 not because he has more privileges for it, he actually has less, but because he has another independent privilege, a driving license.

The crux of the matter is that in the real world there is a distinction between the concept of privilege, or right, and the concept of ability. The ability to perform a certain action may depend on the availability of several privileges(*). In this case: the ability to drive a certain car is formed by the availability of a driving license as well as of the right to drive this particular car. The problem with the access-control scheme is its failure to recognize this difference between privileges and abilities. Under this scheme the availability of a b-ticket with the right r1 in it is sufficient to give a subject the ability to apply the operator P1 to b. Thus, rights are being

(*) In the real world, ability also depends on such things as skill and stamina of the subject, which we have no intention to model.

equated with abilities.

We maintain that to satisfy the principle of attenuation of privileges one must distinguish between privileges and abilities. Such a distinction is being made by the operation-control (OC) scheme to be discussed next. The protection-scheme of the Hydra system [Wul 74] also makes this distinction, but in a less fundamental, and not quite satisfactory way. The Hydra scheme will be discussed in section 5.

4: The Operation-Control (OC) Scheme

Under the OC scheme [Min 77] the ability to perform an operation $P(e_1, \dots, e_k)$ is formed by the availability of two kinds of privileges: a privilege with respect to the operator P , and compatible privileges(*) with respect to the objects e_1, \dots, e_k which serve as the operands to P . Privileges with respect to objects are represented by means of tickets, just as under the AC scheme. However, to represent privileges with respect to operators the OC scheme is using a new device called activator. In this paper only a simplified version of the activator is described.

An activator is a $(k+1)$ tuple

$$\langle P, c_1, \dots, c_k \rangle$$

where P is an identifier of a k -ary operator and c_i , for $i=1, \dots, k$, is a condition defined on the i -th operand of P . The conditions c_i are called the operand-patterns of the activator. The existence of such activator in the

(*) The phrase "compatible privileges" will be clarified later.

domain $D(S)$ serves as a permission for S to apply operator P to a sequence of objects q_1, \dots, q_k in $D(S)$, which "match" the respective activation patterns (or satisfy the conditions) c_1, \dots, c_k . As a notational device, we will give a name, say "A", to an activator by writing

$$A = \langle P, c_1, \dots, c_k \rangle$$

When an activator A is used to authorize the operation $p(q_1, \dots, q_k)$ we will say that the activator A is applied to the objects q_1, \dots, q_k , denoting such an application by $A(q_1, \dots, q_k)$.

An operand-pattern has the form (*)

$$[T; R]$$

where T is a type and R is a set of T -rights. This pattern matches (is satisfied by) any ticket $(b; R_1)$ where b is a T -object and R_1 includes R .

In order to illustrate the authorization role of the activators, and their relevance to our subject matter, we now return to example 1, giving it the interpretation of example 2.

Example 2': Let the type T , of example 1, be CAR. Let $rt(CAR) = \{\text{sell}, \text{drive}\}$ where "sell" is intended to be the right to sell a car, and "drive" is intended to be the right to drive it. Let $op(CAR) = \{\text{SELL}, \text{DRIVE}\}$, which represent the corresponding actions on cars. Now, consider the subjects S_1 and S_2 whose domains D_1, D_2 are described in Figure 1. S_1 who owns the car b_1 has the ticket $t_1 = (b_1; \text{sell}, \text{drive})$ for it. However, he has no DRIVE-activator in his domain,

(*) This is a simplified form of the operand-patterns introduced in [Min 77].

$D2 = \text{domain}(S2)$

$\langle \text{SELL}, [\text{CAR}; \text{sell}] \rangle$

$D1 = \text{domain}(S1)$

$\langle \text{SELL}, [\text{CAR}; \text{sell}] \rangle$

$\langle \text{DRIVE}, [\text{CAR}; \text{drive}] \rangle$

$(b1; \text{sell}, \text{drive})$

$(b1; \text{drive})$

Fig 1

representing the fact that S1 does not have a driving license. Thus, S1 is unable to drive his own car although he has the "drive" right with respect to it. This fact does not make the "drive" right useless. It can be used by S1 to authorize somebody else, S2 in this case, to drive his car. This is done by giving S2 a derivative $t1=(b1;drive)$ of his ticket $t1$. S2 who has the DRIVE-activator $\langle DRIVE,[CAR;drive] \rangle$, representing a driving license, would now be able to drive the car $b1$. Thus the requirements of example 1 are satisfied, without amplification.

Note also that although both subjects have the SELL-activator $\langle SELL,[CAR;sell] \rangle$, which means that both are allowed in principle to sell cars, the driver S2 is unable to sell the car $b1$ because his $b1$ -ticket does not contain the "sell" right, which is required by his SELL-activator [End-of-example].

Just as there may be several different b-tickets which are to represent different privileges with respect to a given object b , we allow for several different P-activators which represent different privileges with respect to a given operator P (see example 3). We now introduce a partial order between activators.

Let A be an activator of order k (with k operand-patterns). We define range(A) to be the set of all possible k -tuples (q_1, \dots, q_k) of objects, which can be matched with the corresponding activation-patterns of A .

Let A and A' be two P-activators for a given operator P . We will say that A' is weaker than A (or, equivalently, A is stronger than A') if range(A) includes range(A'). Such an A' is also called a reduction of A .

clearly, the relation weaker between activators defines a partial order, which is analogous to the partial order defined by the relation weaker between tickets.

Due to the similarities between tickets and activators we will refer to both kinds of objects by the common name "control-objects", or "cobjects", for short. Every cobject represents privileges with respect to the object addressed by it, which may be either an operator (in the case of an activator) or a "passive object" (in the case of a ticket).

Note that the two types of cobjects play complementary roles in our scheme. Neither a ticket nor an activator alone represent an ability to perform any action. Such an ability is formed by the availability of an activator, and one or more matching tickets. To emphasize this complementarity we will use the following terminology.

Let $D(S)$ be the domain of a given subject S . We will use the phrase "power of S " for the set of activators in $D(S)$, and the phrase "range of S " for the set of tickets in $D(S)$.

Thus, the ability of a subject depends on its range, which defines his access rights to various objects, as well as on his power which defines the kind of operations which he can use.

In spite of the (hopefully) intuitive appeal of these terms they do not mean much without specifying the dynamic behavior of the control objects. This issue is discussed next.

4.1: On the dynamics of the operation-control scheme

Here are the rules which govern the dynamics of objects, which are a generalization of the rules previously formulated for tickets.

Rule 1: An existing objects cannot be modified.

Rule 2: Whenever an object o is created, a object is created for it, to be called the primary o-object. (It would be the primary b-ticket if o is the passive object b, or the primary P-activator, if o is the operator P.)

Rule 3: There is an operator duplicate which, when applied to a object c creates another object c' in some other place in the system. c' is called a direct derivative of c. (By the phrase "derivative of c" we mean direct or indirect derivative of c.)

Rule 4: The derivaive c' of a object c is weaker than or identical to c.

Rule 4 is the principle of attenuation of privileges, now extended to activators. An important corollary of this principle is that a object is stronger than, or identical to, every one of its derivatives. In particular, the primary o-object is the strongest o-object.

Note that the above rules do not define completely the dynamics of our authorization scheme. In particular, one must define the operator "duplicate" and its activators, which is done in [Min 77]. To facilitate the following discussion we will make the simplifying assumption that the set of activator in a given domain is fixed. In other words: the "power" of a subject is assmmed to be fixed while its range may vary. Although this assumption cannot be strictly correct for all the domains in a system, it is likely to be correct in many, if not in most cases (see [Min77]). In particular, the "power" of a procedure is likely to be fixed while its range varies from one invokation to another.

4.2: On the meaning of the right-symbols

The meaning of a given right symbol r is best understood in terms of the affect that the absence of r from a ticket t has on one's ability to apply operators to the object addressed by the ticket. We will see in section that such effect depends on the principle of attenuation. To facilitate our discussion we start with an example.

Example 3: Sharing memoranda

Let MEMO be a type of objects which carry memoranda in an information system. Let

$op(MEMO) = \{READ, ADD, DELETE\}$

$rt(MEMO) = \{d, r1, r2\}$

Let the following be the primary activators of the MEMO-operators:

$\langle READ, [MEMO] \rangle$

$\langle UPDATE, [MEMO], [TEXT] \rangle^*$

$\langle DELETE, [MEMO; d] \rangle$

Note that the rights $r1$ and $r2$ are not used by any of these activators.

The significance of this will be clarified later.

(*) The second operand of the operator UPDATE should be a TEXT-object which serves to update the memo.

Now, consider a group of subjects $G=\{S,S_1,S_2,S_3\}$ who are working on the same project but have different responsibilities and authority. They communicate with each other by means of pool of memoranda whose tickets are contained on a file f . A ticket of a memo may be stored in f by members of the group G as well as by other subjects in the system. All the subjects in G are to be allowed to read all the memos on f (namely, all the memos whose tickets are stored on f). However, not every subject is to be allowed to update all documents, or to delete them. Figure 2 gives part of the domains of the subjects in G as well as a sample of the file f . The description of the domain in Figure 2 is incomplete in the following sense. We did not try to account for the assumed ability of the four subjects in G to read the file f and write into it. The ability to read f means that every one of the subjects in G can get MEMO-tickets from f into his own domain. (This ability can be formulated by means of the complete OC scheme [Min77]).

Now, note that all the subjects of G have a copy of the primary READ-activator which enables them to read all memos on f . However they have different versions of the UPDATE-activator: The UPDATE-activator of S does not require any rights in the MEMO-ticket it is applied to. Thus, S can update all the memos on f . On the other hand, the UPDATE-activator of S_1 can be applied only to tickets with the right r_1 , which means that, given the current content of f , S_1 can update only the memos m_1, m_2, m_3, m_4 . Similarly, S_2 can update only m_2 and m_3 , which have the right r_2 . Finally, S_3 whose UPDATE activator requires the rights r_1 and r_2 , can update only the memo m_3 . Note that m cannot be updated by anybody in G .

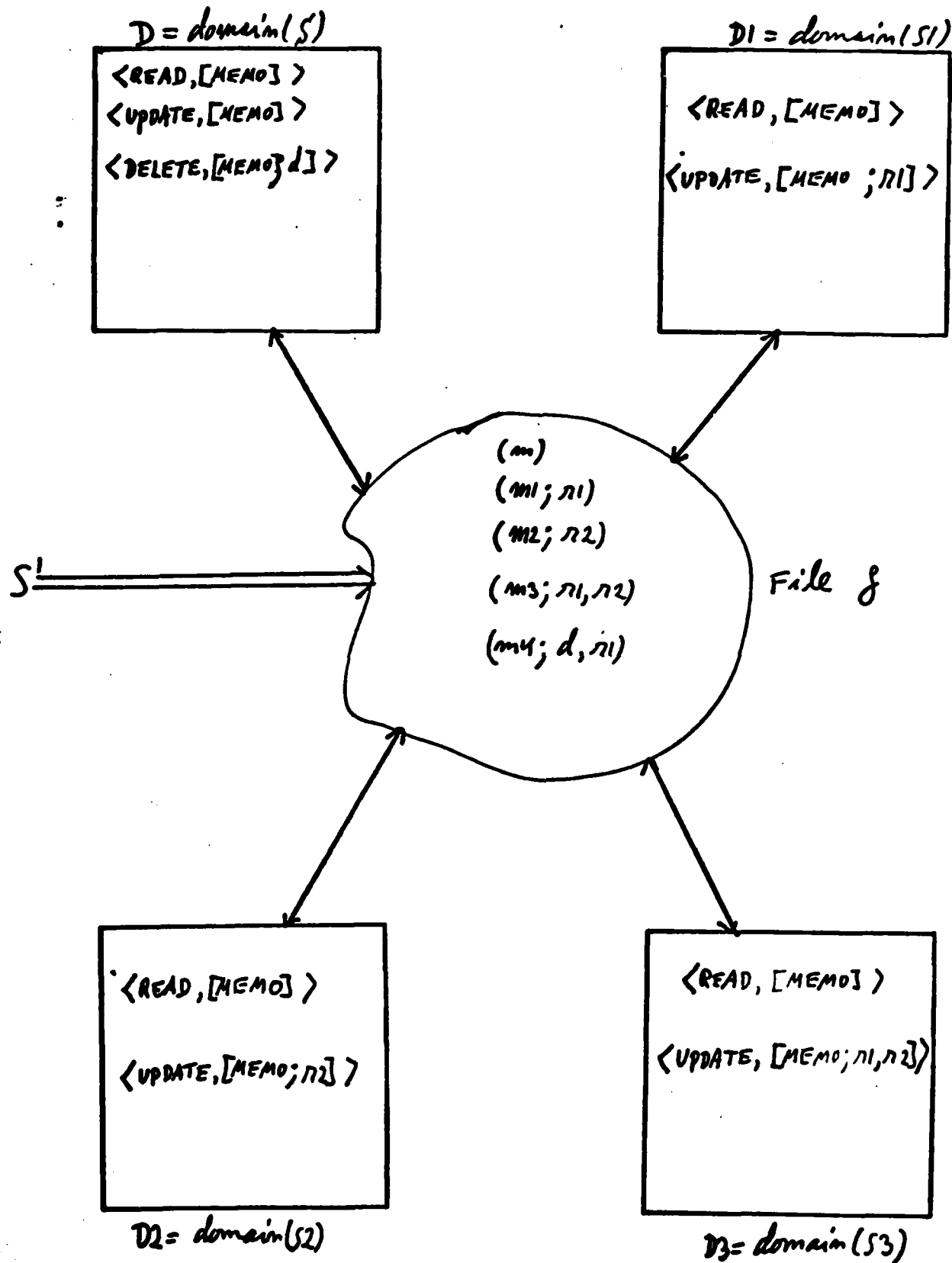


Figure 2

Also, only S has a DELETE activator, so that he is the only one who can delete memoranda. Not all of them, however. Given the content of the file f described in Figure 2, S can delete only m^4 whose ticket has the d right.

Thus, the four subjects in group G have different "power" with respect to the MEMO-tickets, which is formed by the activators in their domains. Knowing about the power of the various subjects in G, the originator S' of a memo can control its disposition, as follows: When S' creates a new memo m' , he gets the primary ticket $t=(m';d,r1,r2)$ for it. If he wants to allow every body in group G to update m' , but he does not want them to delete it, he would insert the reduction $(m';r1,r2)$ of t in the file f. If he wants only S and $S1$ to update m' , then he would insert $(m';r1)$ into f, etc. [end-of-example].

The main purpose of this example is to demonstrate that a right-symbol might have different meaning for different subjects, which allows our four subjects to share the same MEMO-tickets and still have different abilities with respect to the memos. In order to formalize this phenomena we will define the concept of the "authority content", or just "authority" of a right-symbol. In fact, two related concepts of authority- content will be defined.

Definition: The absolute authority content of a given right-symbol r, to be denoted by $U(r)$, is the set of operators whose primary-activators require r.

For instance, in example 3 $U(d) = \{\text{DELETE}\}$.

The significance of $U(r)$ is due to the principle of attenuation: First, because of the attenuation of activators, if the primary P-activator, for a given operator P, has a pattern which requires r, then all P-activators in the system would require r. Thus, the absence of r from a ticket t inhibits the application of P to this ticket. Moreover, because of the attenuation of

tickets, if t does not contain r then no derivative of t would contain r . Therefore, we can make the following statement:

The absence of a right r from a ticket t inhibits the application of operators in $U(r)$ to t itself as well as to all its derivatives.

For example, consider a MEMO-object m and an m -ticket t which does not have the right d . One can put t in the public domain and remain confident that this ticket cannot be used, neither directly or indirectly, to delete the object m .

Note that by the above definition the MEMO-rights r_1, r_2 contain no authority. Indeed, the absence of r_1 from a MEMO-ticket t cannot inhibit the application of any MEMO-operator to it. Yet, it is the existence of r_1 which allows S_1 to apply APPEND to a ticket. In order to cover this phenomena we now introduce the concept of relative authority content of a right-symbol.

Definition: The authority of a right symbol r , relative to a domain D , to be denoted by $U(r/D)$, is the set of operators for which there is a an activator in D which requires r .

For example, $U(r_1/D_1) = \{\text{APPEND}\}$ although $U(r_1)$ is empty. The meaning of the concept of relative authority is summerized by the following statement:

The absence of r from a ticket $t = (b; R)$ in $D = \text{domain}(S)$ inhibits S from applying the operators in the set $U(r/D)$ to t and to its derivatives.

To understand the significance of this statement note the following: First, suppose that P belongs to $U(r/D)$ but not to $U(r)$. Let $t = (b; R)$ be a ticket in $D = \text{domain}(S)$ which does not contain the right r . It is clear that S himself cannot apply P to t . However, S may be able to cause the application of P to the object b , by giving a derivative of t to some other domain D' such that P is not contained in $U(r/D')$.

Secondly, consider an operator P which belongs to $U(r)$ but not to $U(r/D)$ (which can happen only if D contains no P -activators). Obviously, the absence of r from a ticket $t=(b;R)$ in $D=\text{domain}(S)$ has no effect on the ability of S to apply P to object b , because anyway D has no P -activators. However, this absence does prevent S from causing the application of P to b by giving a derivative of t to a subject which does have a P -activator. For example, suppose that the domain $D1$ of example 3 contains the tickets $t1=(m1;d)$, $t2=(m2)$. Since $U(d/D1)$ is empty, the right d does not provide $S1$ with any direct ability. Indeed, exactly the same set of operators can be applied by $S1$ to $t1$ and $t2$. However, $S1$ can enable S to delete $m1$, by placing $t1$ in the file f , which he cannot do for $m2$.

In conclusion note that the OC scheme makes a much more versatile use of the right-symbols than is possible under the AC scheme. One of the benefits which accrue from this use is a more economical utilization of tickets. In example 3, for instance, all the subjects in G were able to use the same directory file. To implement a similar authority structure under the AC scheme one would need four separate directory files, which would result in larger number of tickets and additional complexity in their distribution (see also [Min 77, Section 4]).

5: Amplification in Hydra

Although the authorization scheme of the Hydra system [Wul 74, coh 75] is usually considered to be an access-control scheme, it differs from the AC-scheme outlined in section 2 in one important way: Under Hydra, the availability of a ticket (capability) for an object is not, by itself, sufficient for the

application of an operator to it. One must also have the permission to call the operator. This permission is represented by a ticket for the operator with the "call" right in it. Thus, in Hydra the ability to apply an operator P to an object b is formed by the availability of two kinds of privileges: a privilege with respect to the object b and a privilege with respect to the operator P . However, Hydra recognizes just one type of privilege with respect to an operator, an unqualified right to call it which is represented by a ticket $(P; \text{call})$ with the "call" right. This is to be compared with the varying degree of privileges which can be represented by our activators. This high resolution of privileges with respect to operators, apart of its general usefulness demonstrated by example 3, turns out to be necessary if the authorization scheme is to satisfy the principle of attenuation of privileges. Indeed, the designers of Hydra were forced to violate this principle by the introduction of amplification.

Recognizing the harmful effects of amplification Hydra restricted its use to the subsystems which implement type-extension [coh 75]. Unfortunately, this restriction is both too strict and not sufficient. More specifically we argue that:

- a) The restricted use of amplification in Hydra leaves a class of policies unsupported.
- b) The Hydra's restriction on the use of amplification does not eliminate all its harmful effects.

The second claim will be discussed in the next section where we also show that under the OC scheme type-extension can be implemented without amplifications. To substantiate our first claim let us return to example 2'. We will show that although the authority structure of example 2 'can be supported in Hydra,

without amplification, a simple modification of it cannot be so supported.

The owner-driver situation of example 2' can be implemented in Hydra simply by representing a driving license by a ticket (DRIVE;call) for the operator DRIVE. This ticket should be given to the driver S2 but not to S1.

What makes this case manageable in Hydra is that S1 is not allowed to drive any car. But what about a situation where S1 is allowed to drive some car, but not his own? For example, suppose that S1 is a disabled person who needs a special permission to drive any specific car, permission which may be based on the safety features of that car.

The Hydra solution for example 2' does not work here because both subjects must have the right to call the operator DRIVE. This shows that amplification is necessary in Hydra outside of the modules which implement type-extension.

Under the OC-scheme this authority structure can be represented as follows: Supposed that S1 has the following DRIVE-activator

<DRIVE, [CAR;drive,sd]>

which means that S1 can drive only such a car c' for which it has a ticket (c';drive,sd...), "sd" being this special driving permission. If S1 does not have the "sd" right for his own car c he cannot drive it. The driver, S2, on the other hand, not being disabled, has the more powerful DRIVE-activator <DRIVE,[CAR;d]> which can be used to drive the car c.

Finally, it should be pointed out that there is a way to simulate in Hydra the limited version of the OC scheme described in this paper, as follows: Given a set of P-activators for a given operator P, under the OC scheme, one constructs in Hydra a set of operators whose body is identical to P. These operators differ from each other only in their formal-parameter-specifications

which must be identical to the operand-patterns of the activators which they simulate. The tickets ($P'; \text{call}$) for these operators represent varying degrees of privileges with respect to the original operator P , just like our activators. Indeed, if this is done systematically in a system like Hydra, no amplification would be necessary.

6: The issue of type-extension

In this section we show how type-extension can be achieved under the OC-scheme, without violating the principle of attenuation. Moreover, we will argue that the Hydra implementation of type-extension has some drawbacks due to the amplification on which it is based. But first let us define the concept of type-extension. Definition: Consider a type T' such that $\text{op}(T') = \{Q_i\}$ and $\text{rt}(T') = \{s_i\}$. Let T be a type with $\text{OP}(T) = \{P_i\}$, T is called an extension of T' if the following conditions are satisfied:

- 1) Every T -object is also a T' -object. (Note that this violates the assumption made in section 2 that every object belongs to a unique type).
- 2) The only subjects which are able to apply T' -operators to a T -object are the T -operators P_i . (Thus, outside of the T -operators the only way to manipulate and observe T -objects is indirectly by means of the T -operators which have the exclusive ability to "see the bare representation of T -objects".
- 3) The set of T -rights is

$$\text{rt}(T) = \text{rt}(T') \cup \{r_i\}$$

We will refer to T' as the representation type of T . The set $\text{OP}(T')$ are called the representation operators. Note that every T -ticket may carry T' -rights, to be called representation-rights, as well as the symbols r_i which we call the intrinsic T -rights.

In Hydra [coh 75], all extended types are extensions of a single primitive type $T' = \text{SEGMENT}$. The SEGMENT operators in Hydra are called "generic operators," which includes such operators as GETDATA and PUTDATA. The rights $rt(\text{SEGMENT})$ are called "generic rights". The module which contains the definition of $OP(T)$ and $rt(T)$, for a given type T , is called the T -subsystem.

The main difficulty in the implementation of type-extension is requirement (2) in the definition above. Here is how the designers of Hydra see this problem ([coh 75] p147)

"Hydra must somehow guarantee that ordinary users cannot access or manipulate an object's representation...This implies that ordinary users do not have capabilities [tickets] containing the various generic rights...Yet a subsystem procedure must be able to gain these rights when a capability for an object of the type it supports is passed to it as an argument" (The underlying is mine).

Hydra's solution to this dilemma is an exclusive ability of the representation-operators of a type T to amplify T -tickets (or capabilities, in Hydra's terminology) by adding to them desired representation (generic) rights. We will argue later that even this restricted use of amplification is harmful, but first we will show that under the OC scheme amplification is not necessary for type extension.

6.1: The implementation of type-extension under the OC scheme

Consider a type T' which, like the type SEGMENT in Hydra, serves as a representation-type for a set of extended types. Let

$$AQ = \langle Q, [T'] \rangle$$

be the primary Q-activator of one of the T'-operators Q. AQ is very powerful as it can be applied to any T'-object, regardless of the extended type it hosts. We assume, however, that for every T'-operator Q the primary Q-activator exist only in the module which generates new type-subsystems. Only reductions of these activators are distributed among the various type-subsystems, as follows:

Let T be an extension of T' and let P be a T-operator which needs to apply a representation-operator Qi to its argument. We insert in the domain of P the following reduction of AQi:

$$AQT = \langle Q, [T] \rangle$$

AQT is weaker than AQ because it can be applied only to T-objects. Namely, to a T'-object which hosts a T-object.

Now, since the activators AQT1 exist only in the domains of T-operators, only these operators can apply T'-operators to T-objects, as is required by the definition of type-extension. The dilemma which forced Hydra to use amplification does not exist here since ordinary users do not have any representation activators.

Of course, the invocation of the T-operators themselves is controlled by their own activators. For example, the primary P-activator for a T-operator P may be:

$$\langle P, [T; r] \rangle$$

where r is one of the T-rights. This means that one needs the right r in a ticket in order to apply P to it. In general, the authority to apply T-operators to a T-object can be represented by the intrinsic T-rights {ri}.

Note that this suggests that there is no need to have the representation rights $\{s\}$ in T-tickets. Because, the only way for ordinary users to cause the invocation of T'-operators is by invoking T-operators, and such invocation can be controlled by the intrinsic T-rights. And yet, as we will see below, there is a role to be played by the representation rights in T-tickets.

6.2: The role of the representation-rights

Consider a T-operator P whose primary activator is $\langle P, [T; r] \rangle$. Suppose that P is designed to use a certain T'-operator Q only on some of its invocations, under some special circumstances, say. In this case one may want to allow a subject S to apply P to a T-object b , but only under the condition that such an application would not cause the application Q to b . How can that be accomplished? The problem is that the rights r_i can only control the ability of a subject to apply T-operators to b , they cannot control the internal operation of the T-operators themselves. We propose the following solution to this problem.

Recall that previously we assumed that a T-operator P would have in its domain an activator $\langle Q, [T] \rangle$ for every T'-operator Q which he might have to use. We now suggest that if an operator Q does not have to be used by P on every invocation, then P should have the following activator for it

$$\langle Q, [T; s] \rangle$$

where s belongs to $rt(T')$. This means that P cannot apply Q to its argument unless he has a ticket for it with the right s in it. Therefore, if the subject S has a ticket $t = (b; r)$ he can apply P to it, but this application can not result in the application of Q to b because t does not contain s . If however, we do

not wish to prevent the application of Q to b to result from the application of P to it, then we should give S a ticket $t'=(b;r,s)$, although the T'-right s is not explicitly required by the P-activator.

Thus, the T'-rights can be used in the tickets of T-object as a means to control the internal operation of T-operators. The absence of such a right can prevent a T-operator from applying a certain T'-operator to its argument.

Note that this important use of the representation-rights would be disabled by amplification. If an operator P has the ability to amplify its argument by adding representation-rights to it, as is the case in Hydra, then it does no matter if the operand ticket did not have such a right originally.

It should be pointed out that the designers of Hydra recognized this problem, but only with respect to certain representation-operators (see [coh 75] p152). Indeed, their solution has been effectively, to cancel the amplification for the representation-rights which control these operators. However, they failed to see the more general nature of the problem which requires the complete elimination of amplification.

7: Conclusion

- a) Summery . . .
- b) Other aspects of the OC scheme . . .
- c) Implications of the principle of attenuation to the formal protection models introduced by Harrison, Lipton and others

. . .

REFERENCES

- [Coh 75] COHEN, E.; AND JEFFERSON, D.
"Protection in the Hydra operating system," in Proc. Fifth ACM Symposium on Operating Systems Principles; ACM Operating System Review 9,5, (Nov. 1975), 141-160; ACM, New York, 1975.
- [FAB 68] FABRY, R.S. "Preliminary description of a supervisor for a machine oriented around capabilities," ICR Quarterly Report 18, Univ. of Chicago, Chicago, Ill., 1968.
- [GRA 72] GRAHAM, G.S.; and DENNING, P.J. "Protection--principle and practice," in Proc. 1972 AFIPS Spring Jt. Computer Conf. Vol. 40, AFIPS Press, Montvale, N.J., 1972 p. 417-424.
- [JON 73] JONES, A.J. "Protection in programmed systems," PhD Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., June 1973.
- [LAM 69] LAMPSON, B.W. "Dynamic Protection Structures," in Proc. 1969 AFIPS Fall Jt. Computer Conf., Vol. 35 AFIPS Press, Montvale, N.J., 1969 pp 27-38.
- [LAM 71] LAMPSON, B.W. "Protection," in Proc. Fifth Annual Princeton Conf. on Information Sciences and Systems 1971, pp 437-443. Reprinted in ACM Operating Systems Review (Jan. 1974).
- [LAM 76] LAMPSON, B.W.; and STURGIS, H.E. "Reflections on an operating system design," Comm. ACM 19, 5 (May 1976), 251-266.
- [LIP 77] LIPTON, R.J. and SNYDER, L. "A linear time algorithm for deciding subject security" in J. of the ACM, July 1977 pp. 455-469.
- [LIS 75] LISKOV, B.; and ZILLES, S. "Specification techniques for data abstractions," IEEE Trans. on Software Engineering 1, 1 (March 1975), 7-18.
- [MIN 77] MINSKY, N., "An operation-control scheme for authorization in computer systems," to be published in the Int. J. of Computer and Information Sci. 1978.
- [NEU 75] NEUMANN, P.G.; ROBINSON, L.; LEVITT, K.N.; BOYER, R.S.; and SAXENA, A.R. A provably secure operating system, Stanford Research Inst. Final Report, Menlo Park, Calif., June 1975.
- [RED 74a] REDELL, D.R.; and FABRY, R.S. "Selective revocation of capabilities," IRIA Internat. Workshop on Protection in Operating Systems, Institut de Recherche d'Informa-

tique et D'Automatique, 1974 France, pp. 197-210.

[SAL 75] SALTZER, J.H.; and SCHROEDER, M.D. "The protection of information in computer systems," in Proc. of the IEEE 63, 9 (Sept. 1975), 1278-1308.

[WUL 74a] WULF, W.A.; et al. "HYDRA: the kernel of a multi-processor operating system," Comm. ACM 17, 6 (June 1974), 337-345.

SOSAP-TR-35

August 1977

ON THE USE OF LICENSES AS A PROTECTION MECHANISM

M. Ruschitzka

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.

ABSTRACT

A license is a protection mechanism which is affiliated with an access (algorithm). It specifies by which subject and with what parameters the affiliated algorithm may be invoked. Conceptually, licenses are equivalent to capabilities and access control lists, but many of their properties differ. We discuss several applications of licenses, including network access control and flow control. Licenses may also be used for several of the different checking processes which are typical of contemporary systems. The efficiency of the mechanism is given special consideration, and hardware features for its support are suggested.

Key Words and Phrases: protection, security, access control, flow control, protected subsystems, data types, computer system design.

The ultimate goal of current research in protection and security is the certification of a secure computer system. This goal is approached on several fronts, including system architectures, program certification, protection models, fault tolerance, authentication methods, etc. The ultimate goal may not be achieved for a while, but a number of research systems have been successful with respect to limited security or in locating major obstacles to system security [And,Neu,Pop,Sa4,Wu4]. Since many claims of security have been proven wrong by penetration teams, it is clear that automatic systems must be employed to certify a system [De7,Pop,Wu6]. Several steps are needed to prove a complete implementation correct. These steps include the choice of security assertions, their correspondence to the design of the protection system, the program formulation, the compilation, and the hardware aspects. We are interested in the design aspect of a protection system and how such a system may be supported by the underlying architecture and mechanisms.

It is presumed that the reader is familiar with the basic concepts of access control and the refinements of the standard protection mechanisms [c.f. Sa5]. For the purposes of access control, all information in a system is considered to consist of disjoint objects. Making use of an information object is termed an access, and the active agents performing accesses are defined to be subjects. Accesses to objects occur after being requested by a

subject, and we shall use the triple $\langle \text{subject}, \text{object}, \text{access} \rangle$ to denote such a request. The task of a protection system is to prevent all unauthorized requests from being performed. For this purpose, the protection system maintains a data base which specifies for each request whether it is authorized or not. Lampson [La1] has modeled this data base as an access matrix A such that columns are labeled by object names j and rows are labeled by subject names i . Element $A[i,j]$ of the access matrix contains the names of all authorized accesses which subject i may perform on object j . Since the access matrix tends to be sparse, more efficient ways of encoding the protection data base are required. Two ways of encoding, namely those resulting from compaction of rows and columns respectively, have attained widespread use. After considerable refinements, they evolved into today's capability and access control list schemes.

Both the capability and the access control list scheme have intrinsic advantages and disadvantages. In fact, they appear to complement each other in this respect. Capabilities have a simple intuitive interpretation ("tickets"), can be checked efficiently, and allow flexibility in their manipulation (storing, copying, passing). They pose problems with respect to revocation, propagation, access review, and accumulation. Access control lists, on the other hand, are quite suitable for auditing, access review, and revocation. They suffer from

limited efficiency, allocation problems, and complexity in the search during the check. Many of the problems of both schemes have been alleviated [Red,Sa4], but often at the cost of losing some of the intrinsic advantages. Capabilities tend to excel when they are exercised very frequently while the opposite is true for access control lists. This property can be exploited by using both schemes in the same system (e.g. Multics [Sa4]), but special attention must be paid to conversion problems.

The access matrix can only be compacted in two dimensions, but the protection data base may be represented by two other matrices, the object matrix and the subject matrix. Either one of them is labeled by access names along one dimension, thus suggesting a third protection scheme in which the protection data are distributed over accesses. (The term algorithm will be used interchangeably henceforth.) The basic datum affiliated with an algorithm is a <subject; object> pair; we refer to it as a license. The subject specifies an authorized requestor, the object specifies the parameter which this requestor may use for the algorithm. Multiple parameters are acknowledged by replacing the term object by the term object-list. Note that this allows the authorization of the composite request COPY(B,A) without necessitating the authorizations of the requests GET(A) and PUT(B) for the same subject. Like capabilities and access control lists, licenses offer advantages and problems. A decade ago, when the only access

attributes of interest were READ, WRITE, and EXECUTE, licenses would not have had much appeal since the degree of distribution of the protection data base would have been negligible. However, with data types playing an increasingly important role and with program verification techniques maturing, the license mechanism offers several novel features for the support of sophisticated protection systems.

LICENSES

A license is a <subject; object-list> pair affiliated with an algorithm. It authorizes the specified subject to invoke the affiliated algorithm with the specified object-list as parameters. At any one instant in time, the set of licenses in the system completely specifies all currently authorized operations. Although the licenses affiliated with a particular algorithm could always be expressed as a set of individual <subject; object-list> pairs, efficiency considerations will often dictate a compressed encoding of the protection state. For instance, if the object-lists of a large number of licenses are identical, these licenses may be replaced by a generalized license with the same object-list, but with an appropriate subject class identifier in the subject field. The compression of licenses is quite similar to those techniques which are used to compress access control lists (c.f. [Sa4]).

From a user's point of view, the protection mechanism follows the conventional notion of licensing. Two aspects of licensing are crucial. First, licenses must be issued by an appropriate authority. Second, before the licensee (the operator) may proceed to act upon a request, it must be checked that the request is authorized by the license issued to the licensee. This check of the request (the mediation) may be performed by the licensee or by some other trusted agency. Driver licenses and liquor licenses are typically mediated by the licensee, but airline pilot licenses are also thoroughly mediated by the airline for which a pilot flies. While a license intrinsically implies a right, it also defines this right quite narrowly. A liquor license implies the right of selling liquor, but it also limits this right to certain types of liquor, certain ages of customers, certain hours of the day spelled out by state laws, etc.

As a protection mechanism, a license is issued to (affiliated with) an algorithm by consensus of the controllers of all quantities involved in a request. In particular, the consensus of the controllers of the algorithm, of all objects in the object list, and of all subjects listed in the license, are required. The verification of the consensus and the attachment of the corresponding license(s) to an algorithm is referred to as licensing. Licensing of algorithms is performed by the protection system which stores the new license(s) in the license-part of the algorithm.

When the execution of an algorithm is requested, the identifier of the requesting subject and the parameters (identifiers of all objects in the object-list and parameter values) are compared with the license-part of the algorithm. A protection fault occurs if no match is found. We delay discussion of the format of licenses since it depends strongly on the type of algorithm with which a license is affiliated. Algorithms are defined in terms of other (lower-level) algorithms and efficiency considerations demand that the complexities of an algorithm and the corresponding mediation mechanism are somewhat related. In particular, the overhead involved in mediation should be small compared to the execution of the algorithm. Thus, licenses of hardware algorithms will often consist of only a few bits, while the licenses affiliated with a complex procedure may well occupy an entire secondary storage block.

THE TECHNOLOGICAL HIERARCHY OF ALGORITHMS

Algorithms are static descriptions of clerical procedures which may be invoked for execution or interpretation. We shall refer to the interpretation of an algorithm as an operation. The form of the static description depends on the complexity of the algorithm and may be implemented in hardware or software. An algorithm for a register-register transfer will often be implemented as a micro instruction; the static description is then provided by a specific gate configuration. Conversely, the

algorithm for a compilation is typically represented by a procedure, a sequence of machine instructions. In general, we may conceive of algorithmic descriptions as a partial ordering with the relation "is a sequence of". This partial ordering defines an algorithmic hierarchy: commands, procedures, machine instructions, [micro instructions, [nano instructions,]] logic control lines. Applying the concept of levels of abstractions [Dij], procedures may further be subdivided into different levels of procedures. (We defer for the time being the case of a procedure calling another one of the same level.) Note that every algorithm in the system is uniquely identified by a level number and an algorithm number of that level. (The algorithm number of a machine instruction is simply the operation code.) The algorithmic hierarchy is depicted in Figure 1. Other aspects of this figure will be discussed below.

An operation, i.e. an interpretation of an algorithm, proceeds according to the static description of that algorithm. Every system provides the necessary controls for fetching the algorithm number and the parameters of the algorithm to be invoked next, and for executing it. A different interpreter (level interpreter) is provided for this purpose on each level of the hierarchy. In particular, control lines are interpreted by gates, nano instructions by the nano control unit, micro instructions by the micro control unit, machine instructions by the control unit, and commands by the command level interpreter (command

<u>ALGORITHM</u>	<u>LEVEL INTERPRETER</u>	<u>DATA BASE</u>
Command	Command Level	Secondary Storage
Procedure	Interpreter	
(several levels of abstraction)	Operating System Modules (scheduler, CALL, interrupt system)	
Procedure		Primary Memory or Registers
Instruction	Control Unit	Registers
Micro Instruction	Micro Control Unit	
Nano Instruction	Nano Control Unit	
Logic Control Line	Gates	

Fig.1. Correspondence between the hierarchies of algorithms,
level interpreters, and protection data base media.

processor). Portions of the operating system, like the scheduler, the interrupt system, and the CALL, interpret procedures on the various levels. Note that each level interpreter is itself a sequence of lower level algorithms. For example, the command level interpreter is a sequence of procedures and the control unit (unless it is hard-wired) is implemented as a sequence of micro instructions.

The protection mechanism for mediating an operation is closely related to its corresponding level interpreter. The level interpreter of any level loops over the two phases

fetch; execute;

thus controlling the sequence of consecutive operations. Mediation may be performed between these two phases:

fetch; mediate; execute;

Mediation involves a comparison of the identifier of the currently active subject and the parameters (object-list) with the license(s) affiliated with the algorithm. The overhead is a function of both the type of comparison (simple match, associative search) and the access time of the storage in which the licenses reside. For a command, an associative search of licenses residing on secondary storage is satisfactory since most likely the command itself must be paged in. For a machine instruction, however, all quantities involved in the mediation must be in high speed storage or at least in primary memory. In the next section we shall discuss various encodings of licenses on different levels as well as their lifetimes.

The implication of a verified algorithm on the form of level interpreters deserves some attention. The current state of the art allows the verification of short procedures, but fails with respect to complex software. On the other hand, hardware algorithms can readily be verified by standard testing techniques. Thus, the form of a hardware level interpreter

fetch; mediate; execute;

is typically replaced by

fetch; mediate while executing;

This form is more efficient since the mediation does not have to apply to all algorithms of the particular level, but may be specialized for one or more algorithms. Privileged instructions in which the mode is checked during the execution of the instruction represent a trivial example of this specialization. The Multics access checking logic for the implementation of rings in hardware represents a more involved example [Sh2]. Similarly, procedures of low levels of abstraction which tend to be more easily verified, may be trusted to perform their own mediation. On a higher level, both forms may coexist by slightly modifying the corresponding level interpreters:

fetch; if verified mediate while executing

else mediate before executing;

For efficiency reasons the protection checking mechanism in capability systems is tightly coupled with the addressing mechanism [Sa5]. Quite analogously, the license

mechanisms for mediating requests are closely coupled with level interpreters. It follows that these mechanisms are distributed over the algorithmic hierarchy and can be designed to minimize the cost/performance characteristics of the protection system. Furthermore, while efficiency considerations are used in the design of the mechanism of a particular level, the conceptual framework applies to all levels independent of their technological characteristics. Note that the same mechanisms are used for access control and parameter checking; the latter aspect has proven particularly troublesome in attempting to verify commercially available systems. In all protection systems the CALL governing transitions between procedures plays an important role. The notion of level interpreters generalizes this primitive for all types of algorithms.

THE HIERARCHY OF LICENSES

In a pure licensing scheme, the protection state, i.e. the collection of triples encoding authorized operations, is distributed over the algorithmic repertoire of the system. Dynamic mediation is performed when an operation is invoked by a level interpreter. In order to keep the overhead tolerable, the access times of all quantities involved in the mediation must be comparable to the fetch time of the algorithmic specification. When a procedure must be fetched from secondary storage, it is sufficient to have its licenses resident on secondary storage. On the other hand,

the licenses of a machine instruction must be readily accessible in high speed storage. Consequently the protection state itself is distributed over a hierarchy of storage media which is related to the algorithmic hierarchy. Figure 1 illustrates. We refer to the licenses of one algorithmic level as the (protection) data base of that level. Conceptually, the data base of a level is organized around the algorithms of that level, i.e. <subject; object-list> pairs are grouped and affiliated with their respective algorithms. In an implementation, algorithms of the same level may often have identical licenses affiliated and duplication of the contents of their license-parts may be avoided by merging them.

Some of the existing hardware protection features may be viewed in this light. The processor state bit is an example of a (rudimentary) subject class identifier used on the machine instruction level. Here all subjects are divided into two groups (user and system). Let us also assume for the moment that the object-list field in all licenses will match any parameters. Thus, the set of all possible licenses reduces to the following three: <user; ALL>, <system; ALL>, <user or system; ALL>. Accordingly, all machine instructions may be partitioned into three groups each of which shares an identical license. In a hardwired control unit these licenses are wired into the instructions and mediation involves a simple comparison with the processor state bit. n-state architectures with $n > 2$,

like ring structures, allow for further discrimination among subjects.

A standard paging map as depicted in Figure 2 may be considered part of a micro instruction data base. It contains the licenses for the micro instructions read, write, execute (i.e. read during the fetch phase of the machine instruction interpreter), and map. Objects are identified by virtual page numbers. The license of read, for example, is of the form <ALL; any virtual page number with R-bit set>. The column of invalid bits forms the license of the micro instruction map which performs the mapping of virtual into physical page numbers. In these examples of existing hardware protection features the object-list fields in machine instruction licenses are degenerate and so are the subject fields of the licenses of the four micro instructions. Degenerate licenses are more efficient in the mediation process, but more complex formats are justified in the case of more complex algorithms like operating system support instructions.

The hardware examples served to demonstrate the necessity of updating protection data bases at lower levels dynamically in order to keep the cost (in terms of both complexity and storage) of the mediation process reasonable. We generalize this notion to all levels. Upon successful mediation of an operation on one level, licenses are converted and added to the data base of the next lower

virt. page no.	R	W	E	I	phys. page no.
0					
1					
2					
n-1					

R, read bit; W, write bit; E, execute bit; I, invalid bit.

Fig.2. A standard paging map viewed as a micro instruction data base.

level. (In implementations, depending on the nature of a particular system, a higher level interpreter may cause the data bases of several lower levels to be updated with licenses.) In essence, the new licenses on the lower level are derived from the licenses involved in the mediation process. The newly created licenses on the lower level reside there until the operation responsible for their creation terminates. This scheme is motivated primarily by the desire to avoid the longevity problem of capabilities with all its consequences while retaining most of their efficiency characteristics.

Procedures and commands differ from lower level algorithms in that they may be defined in terms of algorithms of the same level. This does not change the basic scheme for creating and deleting licenses at the procedure levels. Similarly to the CALL stack employed in contemporary operating systems for control purposes, a stack is used to link the saved licenses of the data bases of procedures and lower levels when control passes from one procedure to another of the same level. Upon RETURN from a procedure, the licenses in the top element of the stack are deleted. It is instructive to compare this method with the handling of templates in HYDRA [Coh]. Before a procedure may be invoked, the triple <calling subject; procedure; parameters> must be mediated. In HYDRA, the mediation is split into two independent steps. First, the calling subject must present a capability for executing the procedure.

Second, the parameters (the passed capabilities) are compared with the templates which are affiliated with the procedure. A positive outcome results in the creation of new capabilities for the lifetime of the procedure. Like templates, licenses are affiliated with procedure. The mediation, however, is performed in one step by comparing the licenses with the identifier of the calling subject in the context of the passed parameters.

ON THE ENCODING OF LICENSES

Contemporary hardware architectures offer only minimal support for protection systems. This support includes processor modes, memory maps, and descriptor mechanisms. However, even tag bits for capabilities or capability registers are rare (c.f. [Eng] as an example). In fact, most commercially available processors lack the necessary features to aid in the implementation of a secure system [Sm1]. The increase in cost-effectiveness of hardware features and microcode over software in the past few years has resulted in the enhancement of hardware architectures with operating system support features (e.g. virtual memory mechanisms, special purpose instructions). A similar trend is to be expected for protection features. In particular, we suggest a feature which would allow the labeling of information containers (page frames, register sets, tapes) in a uniform manner across an entire system, including periphery. Any physical storage unit should feature a

string of bits describing its current contents. For a primary memory page this string would be stored in an affiliated register. On secondary storage this string would precede (and/or accompany and/or trail) the corresponding information block. These strings would aid in the design of memory management systems which are so closely related to protection systems that they are often included in the system kernel. The contents of these strings is twofold: a unique identifier of the virtual object contained in the physical storage unit and a semantic description of its contents (e.g. security class, type).

The unique identifiers for algorithms are given by a level number and a unique algorithm number. For procedures, the latter is the procedure segment number which corresponds to a pathname in the procedure directory starting with the name of the controlling user. Operations (invoked algorithms) are identified by the algorithmic path name, i.e. the sequence of algorithms which define their current context. Thus, a machine instruction in execution is identified by the sequence: command identifier - procedure identifier(s) - operation code. Subjects are uniquely identified by the name of the invoking user concatenated with the current operation identifier. Note that part of the subject identifier changes with every instruction. Semantic information about a subject includes a security class and a privilege class. The mapping between virtual and physical objects is maintained by the memory management

system. The contents of any physical location is always part of a virtual object (segment). The mapping of bits, words, pages, and segments defines an object hierarchy which is utilized in the conversion of licenses when they are added to the data base of the next lower level. Segments are identified by segment numbers; they correspond to pathnames in the object directory starting with the name of the controlling user. Semantic information about an object includes a type and a security level.

The design of a protection system based on licenses depends on the concrete specifications of the formats of identifiers (including semantic information). These specifications, in turn, depend as much on the hardware architecture as they depend on the specific operating system structures. We have simulated both a hardware configuration and an operating system to investigate the properties of licenses. Due to its specific nature, our design is beyond the scope of this paper and we refer the reader to [Shi] for details. A general comment about our effort is in order though. The set of licenses affiliated with an algorithm is conceptually a list and there are many analogies to access control lists as far as compression techniques are concerned [c.f. Sa4]. What distinguishes a license from an access control list entry is the fact that mediation involves a context-sensitive comparison of a subject identifier consisting of many subfields with several parameters (especially object identifiers with again many subfields).

The subfields help since often only one of them has to be consulted to make a decision. The context-sensitive nature results in the notion of identifier variables in the licenses. Consider the case of a procedure which the controller would like to make available to all users under the condition that they may pass only objects under their own control. Assigning the variable x to denote any user name, the compressed license can be represented as $\langle x; x.ALL, x.ALL, \dots x.ALL \rangle$. If x denotes the name of the controlling user, the above license specifies a no-sharing situation and illustrates the economy of the scheme for simple policies.

The components of a license do not only contain identifiers, but also semantic information about the involved objects and subjects. When the mediation process can be based on the semantic information alone, it is usually considerably more effective. In the next two sections we discuss several applications for which the license scheme seems to be particularly suited.

DATA TYPES, PROTECTED SUBSYSTEMS AND NETWORKS

A data type defines a class of objects by specifying its internal data representation and the externally invocable access procedures. It is a predominant feature of data types that the internal data may only be accessed via the specified access procedures. The notion of data types evolved in a number of programming languages, like Simula

[Dah], CLU [Lis], and Alphard [Wu6], to support the discipline of structured programming. It has since become an important operating systems notion too [c.f. Coh]. Data types may be created by a user by obtaining a unique type identifier from the system. In doing so, this user becomes the controller of the new type. After the appropriate access procedures have been coded, they are licensed such that a passed parameter specifying the new data type must be of that type. The subject fields in the licenses of the access procedures need not be identical, thus allowing for discrimination among subjects with respect to different accesses to the same type. No further steps are required. Other users may make use of the new type if they have been included in the licenses of the access procedures. Otherwise, they may create data objects of the new type, but cannot access them themselves. An attempt to circumvent this restriction by coding new access procedures would fail at the time of licensing since the consensus of the controller of the new type is required. (This consensus may be given under special circumstances, e.g. when a type controllership is passed from one user to another.) The internal data representation of a new type must be defined in terms of existing types and the descriptor to a data object of this type maps its name into a set of descriptors of objects of the defining types. Note that the access procedures of the new type must be coded in terms of accesses to the defining types, thus necessitating the

consensus of their controllers. A data type is a unique abstract description according to which an arbitrary number of concrete objects may be created. Related to this concept is the concept of a protected subsystem of which only one instance may be created.

A protected subsystem is a collection of procedures and data objects that is encapsulated in such a way that the data objects may be accessed only by the procedures of the protected subsystem. Protected subsystems are instrumental in the implementation of sophisticated security policies. Many cases of such policies have been discussed in the literature, in particular the handling of mutual suspicion [ShD, Sa5], confinement [La9, Sa5], and user-defined access control like access to student grade records [Gra, Sa5].

The basic license mechanism supports the implementation of protected subsystems. Since all procedures have license-parts which specify the subjects that may invoke the procedure with certain parameters (including objects), any procedure is potentially part of a protected subsystem. Suppose data objects D and E are to be encapsulated with procedures P and Q. At the time of creation of P and Q they are licensed to access D and E (i.e. $\langle S; D, E \rangle$ pairs are affiliated with them where S specifies the authorized callers). As long as no other procedure is ever given a license including D or E, the two procedures and the two data objects form a subsystem.

Figure 3 illustrates an example concerning the policy that physicians (PHYS) should have complete access to patients' records while statisticians (STAT) should be constrained by privacy considerations. We assume first that patients' records are kept in two separate data objects, with the personal data stored in object PERS and the medical ones in object MED. The procedure PPHYS is coded to service all physicians' accesses and is licensed with <PHYS; PERS, MED>. Similarly, the procedure PSTAT serves statisticians and is licensed with <STAT; MED>. Note that the licenses do not only determine which data objects may be accessed, but also which subjects may invoke the particular procedure. Of course, the data objects PERS and MED could be merged into a single object. In this case, the procedure PSTAT would have to be verified to ensure that no personal data are returned to statisticians. If statisticians were allowed to obtain averages of personal data, but not the personal data themselves, verification is unavoidable since this case subsumes the general case of data transformations.

The operating systems of most computer systems may be viewed as protected subsystems. User programs rely on accesses to system-maintained data, but these accesses must be performed via (more or less) verified system procedures. Conceptually, the notion of a protected subsystem may be further extended to computer networks. Figure 4 shows a network configuration with several hosts. A message arriving at a host represents a request for a specific

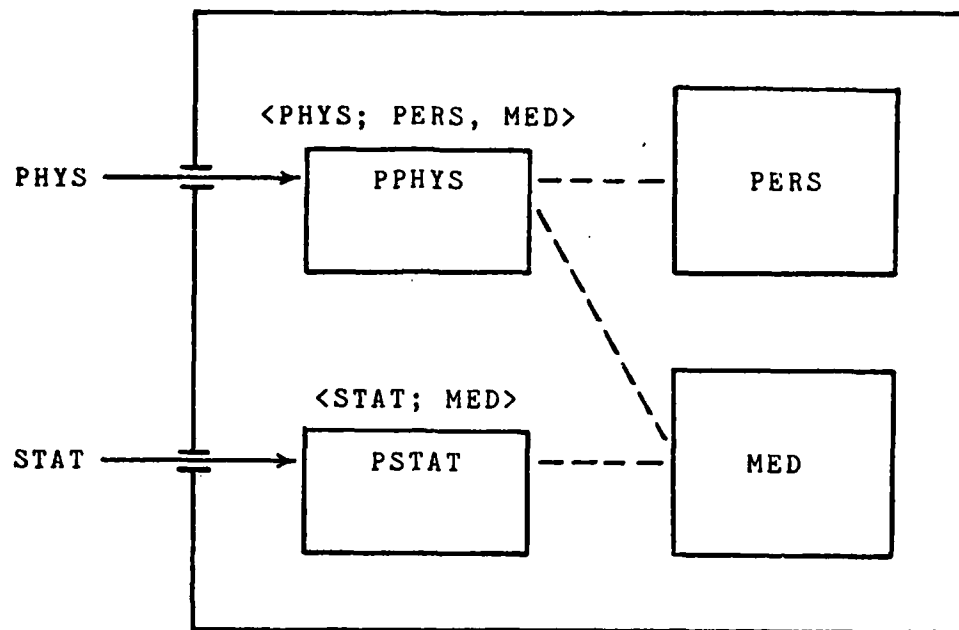


Fig.3. A protected subsystem for a privacy policy.

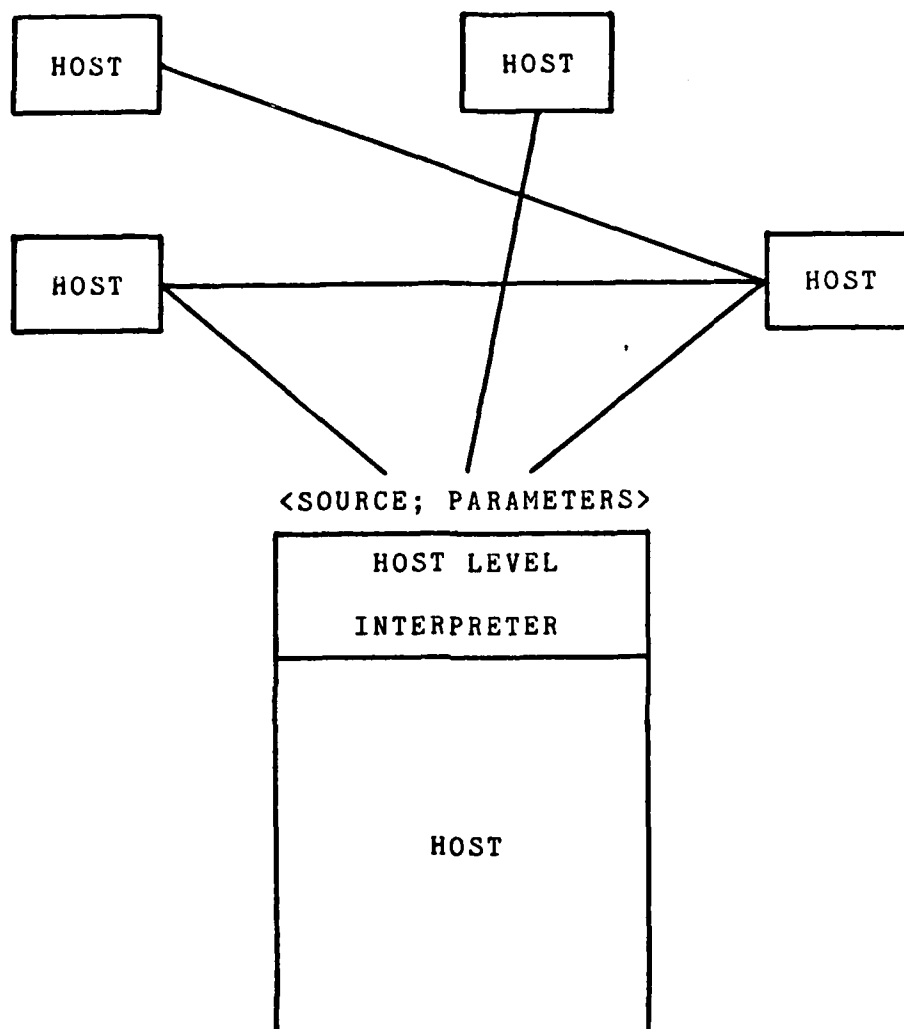


Fig.4. The level interpreter on the network level.

operation on that host. We extend the hierarchy of level interpreters to include a host level interpreter and an affiliated mediation mechanism. This mechanism mediates by matching the relevant message portions (source subject, requested operation, and parameters) with the licenses in the host protection data base. Of these relevant message portions only the identifier of the source subject must be unforgeable. The integrity of this identifier can be assured by a simple and efficient scheme based on Lampson's model of the Message System [La1]. Modifications of other portions of the message en route does not jeopardize the security of the license mechanism; in the worst case, it would result in an unnecessary, but still authorized, operation.

FLOW CONTROL

Access control mechanisms are designed to support the dynamic mediation of requests concerning the accesses of objects by subjects. Authorized requests are denoted in the protection data base. To a large degree, the encoding of this data base determines the nature of the access control mechanism. In the capability scheme, the access control list scheme, and the license scheme, the protection data base is organized along subjects, objects, and algorithms respectively. Owing to the distributed nature of the data base in all three schemes, it is often difficult to account for all possible data movements (with possible

transformations on the way) in a global sense. Flow control concerns itself with this problem. A secure system must provide mechanisms to support the control of both accesses and information flow.

An appealingly simple model for flow control [De6] has resulted in a certification mechanism for verifying information flow through a program [De7]. In this model, objects (and processes) are bound to disjoint security classes which serve to define authorized information flow. Briefly, the result of a procedure which takes data from several input objects may only be stored in an output object of a given security class if information may flow from all security classes to which any input object is bound to the security class of the output object. Note that authorized flow is defined independent of the particular procedure which controls the transformation of the flowing information. This property simplifies the necessary mechanism. It also implies that the security class of derived information must not be less than the security classes of information used in the derivation. On the other hand, a procedure-dependent mechanism would increase the versatility of the model. In particular, it would be possible to decrease the security class of selectively extracted portions of an object. For example, the statistical data extracted from a medical data base should have a lower security class associated than the medical data themselves. Similarly, the output of an encyphering

procedure is intrinsically less sensitive than its input.

For access control, the object-list field of a license already contains an encoding (including type and security class) of all authorized input and output objects. Thus, if the security class of the output object passed by the caller does not match the security classes of the input objects, a protection fault will result. (The same holds true for the case of several output objects.) Thus, the license mechanism requires no changes in order to implement procedure-dependent flow control. The valid range of security classes of output objects is statically determined at the time of licensing. To implement dynamic binding of objects to security classes, we add to the license-part of a procedure a description of the mapping of the security classes of the input objects to those of the output objects. Furthermore, the mediation mechanism affiliated with the procedure level interpreter is enhanced to evaluate this mapping and to update the security classes of the output objects. By default, all procedures would have the standard least upper bound mapping assigned and the consensus of the protection system is required to specify a decreasing security mapping.

Licenses derive many of their characteristics from the fact that object identifiers and the affiliated semantic information are viewed like parameters. This property allows context-sensitive access control with respect to

object combinations, in particular combinations of input and output objects. The same property aids in the implementation of flow control. Thus, licenses appear to be a suitable mechanism for the implementation of both access and flow control.

SIMULATION ASPECTS

We have designed and simulated an integrated hardware and operating system configuration to investigate the properties of a protection system based on licenses. Our design is strongly influenced by an educational system which is being simulated as part of a course in operating systems [RuA]. In this course, special emphasis is put on the interactions between hardware and software, and the major components of the system - the machine architecture (COS [Ru1,Ru2]), the implementation language (CAL [RuL]), and the operating system (COSMOS [RuM]) - have been designed accordingly. While we adopted many architectural features of this educational system, the internal structure was completely redesigned to lend support to the implementation of the protection system.

The simulation is coded in SIMULA [Dah] and documented in project notes [Shi]. The hardware configuration consists of nano programmable and micro programmable central processors, primary memory modules, disk systems (each one consisting of a controller and a drive), and terminals. The various components are represented by SIMULA classes and

their number is parameterized. The virtual address space is segmented and segment numbers are absolute. Segments are referenced via descriptors and are also paged. The major operating system procedures handle initialization, scheduling, CALL's, interrupts, terminal I/O, disk I/O, memory recognition, and command recognition. The protection system consists of the mediation algorithms affiliated with the four level interpreters, the data structures for identifiers and licenses, and the routines for updating these data structures. Various formats of licenses and identifiers have been tried out, but further research is needed in this area.

The low level of the simulation (SIMULA code is used exclusively for the interpretation of nano instructions) has focussed our attention on the lack of hardware support features for the implementation of a protection system. The periphery is particularly weak in this respect. Along these lines, we have begun the design of a protection device which is to be interfaced with the UNIBUS of a PDP-11/60. This device is meant to act as an extension of the protection system (typically located in the CPU) for the purposes of mediating operations on the I/O bus.

CONCLUSION

Licenses are equivalent to capabilities and access control list entries in the sense that each of them is an encoding of the protection state, where the encoding

consists of two of the three quantities subject, access (algorithm), and object (or object-list). In the case of licenses - <subject; object-list> pairs -, the protection state is affiliated with algorithms. Mediation involves a context-sensitive comparison of licenses with the requesting subject and its parameters. Thus, the same mechanism serves for access control and parameter checking. (In special cases, authentication may be viewed as an instance of parameter checking, e.g. LOGIN(user name, password or fingerprint).) Licenses support the implementation of data types and protected subsystems and are particularly suited as a network protection mechanism. Furthermore, they serve as a mechanism for monitoring information flow, thus unifying the concepts of access control and flow control.

The notion of the algorithmic hierarchy serves as a design criterion for the specification of licenses on the various levels. The hierarchical organization of licenses provides a unifying framework which allows efficiency considerations to dictate the design on each level. Some of the existing hardware protection features have been shown to be consistent with the license mechanism. Since the hierarchy blurs the distinction between hardware and software, manageable hardware techniques suggest themselves as remedies to some of the more intricate software problems.

Our simulation has proven the feasibility of implementing licenses in a controlled environment. Further research is needed for the design of an operational system based on this mechanism. In particular, the relation between the hierarchies of objects (bits, fields, words, records, pages, segments) and subjects (processors, processes, computations), which vary considerably from system to system, deserves careful attention. Furthermore, more hardware support will be needed for licenses on the lower levels. In any event, licenses offer several advantageous properties and may readily be applied on the network and command level, possibly in conjunction with capability and/or access control mechanisms on the lower levels.

REFERENCES

- [And] Andrews, G.R. COPS - A mechanism for computer protection. Proc. Workshop on Prot. in Op. Sys. IRIA, Rocquencourt, August 1974, pp. 5-26.
- [Coh] Cohen, E., and Jefferson, D. Protection in the HYDRA Operating System. Proc. Fifth Symp. on Op. Sys. Principles, U. of Texas, Austin, November 1975, pp. 141-160.
- [Dah] Dahl, O.J., and Nygaard, K. Simula - An Algol-Based Simulation Language. Comm. ACM 9, 9 (Sept. 1966), 671-678.
- [De6] Denning, D.E. A Lattice Model of Secure Information Flow. Comm. ACM 19, 5 (May 1976), 236-243.
- [De7] Denning, D.E., and Denning, P.J. Certification of Programs for Secure Information Flow. Comm. ACM 20, 7 (July 1977), 504-513.
- [Dij] Dijkstra, E.W. The Structure of the "THE" - Multiprogramming System. Comm. ACM 11, 5 (May 1968), 341-346.
- [Eng] England, D.M. Capability Concept Mechanism and Structure in System 250. Proc. Workshop on Prot. in Op. Sys. IRIA, Rocquencourt, August 1974, pp. 63-82.
- [Gra] Graham, G.S., and Denning, P.J. Protection - Principles and Practice. Proc. AFIPS 1972 SJCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 417-429.
- [La9] Lampson, B.W. Dynamic Protection Structures. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 27-38.
- [La1] Lampson, B.W. Protection. Proc. Fifth Annual Princeton Conf., Princeton U., March 1971, pp. 437-443.
- [Lis] Liskov, B. An Introduction to CLU. Comp. Str. Group Memo 136, Lab. for Computer Science, M.I.T., Cambridge, Mass., Feb. 1976.
- [Neu] Neumann, P.G. et al. On the design of a provably secure operating system. Proc. Workshop on Prot. in Op. Sys., IRIA, Rocquencourt, August 1974, pp. 161-176.

- [Pop] Popek, G.J., and Kline, C.S. Verifiable Secure Operating System Software. Proc. AFIPS 1974 NCC, Vol. 44, AFIPS Press, Montvale, N.J.
- [Red] Redell, D.D. Naming and Protection in Extendible Operating Systems. PH.D. Diss., U. of California, Berkeley, 1974.
- [RuA] Ruschitzka, M. An Operating Systems Implementation Project for an Undergraduate Course. Proc. Seventh Tech. Symp. on Computer Science Ed., Atlanta, Georgia, Feb. 1977, pp. 77-84.
- [Ru1] Ruschitzka, M. COS-Model 1 Reference Manual. CS 416/18 Class Notes, Dept. of Computer Science, Rutgers U., New Brunswick, N.J., 1977.
- [Ru2] Ruschitzka, M. COS-Model 2 Reference Manual. CS 416/18 Class Notes, Dept. of Computer Science, Rutgers U., New Brunswick, N.J., 1977.
- [RuL] Ruschitzka, M. CAL-CAROL Reference Manual. CS 416/18 Class Notes, Dept. of Computer Science, Rutgers U., New Brunswick, N.J., 1977.
- [RuM] Ruschitzka, M. COSMOS Reference Manual. CS 416/18 Class Notes, Dept. of Computer Science, Rutgers U., New Brunswick, N.J., 1977.
- [Sa4] Saltzer, J.H. Protection and the Control of Information Sharing in Multics. Comm. ACM 17, 7 (July 1974), 388-402.
- [Sa5] Saltzer, J.H., and Schroeder, M.D. The Protection of Information in Computer Systems. Proc. IEEE 63, 9 (Sept. 1975), 1278-1308.
- [ShD] Schroeder, M. Cooperation of mutually suspicious subsystems in a computer utility. Ph.D. Diss., M.I.T., Cambridge, Mass., 1972.
- [Sh2] Schroeder, M.D., and Saltzer, J.H. A hardware architecture for implementing protection rings. Comm. ACM 15, 3 (March 1972), 157-170.
- [Shi] Shiao, M.-D. Secure Operating System. Project Notes, Dept. of Computer Science, Rutgers U., New Brunswick, N.J., 1977.
- [Smi] Smith, L. Architectures for Secure Computer Systems. ESD-TR-75-51, The MITRE Co., Bedford, Mass., April 1975.
- [Wu4] Wulf, W. et al. HYDRA: The Kernel of a Multiprocessor Operating System. Comm. ACM 17, 6

(June 1974), 337-345.

- [Wu6] Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of Alphard programs. IEEE Trans. on Software Eng. SE-2, 4 (Dec. 1976), 253-265.

AN OPERATING SYSTEMS IMPLEMENTATION PROJECT
FOR AN UNDERGRADUATE COURSE

M. Ruschitzka

Department of Computer Science
Rutgers University
New Brunswick, New Jersey

AN OPERATING SYSTEMS IMPLEMENTATION PROJECT
FOR AN UNDERGRADUATE COURSE

Manfred Ruschitzka
Rutgers University
November 1976

ABSTRACT

While the adoption of an implementation project for an operating systems course is certainly beneficial, non-trivial projects are inherently demanding in terms of student efforts and computer costs. This paper reports on a project which has been designed to keep the effort for an extensive simulation of a contemporary system within acceptable limits. The project involves both a hardware simulator and an operating system, and a considerable reduction of the overall effort could be achieved by enhancing the hardware with operating systems support features. The design criteria as well as the characteristics of the resulting hardware configuration and operating system are presented, and the value of the project as a teaching tool is discussed.

INTRODUCTION

Since the introduction of multiprogramming, operating systems have become a respectable discipline within computer science and most computer science programs now include at least one course on their design and implementation. While the theoretical material of such a course is generally agreed upon [1,2], the question of whether or not such a course should stress an implementation project, and what its nature should be, has continued to be a much debated issue [3,4,5,6,7,8]. It is not the value of such a project which is being debated. Clearly, the experience gained from an implementation cannot be substituted by even the most illuminating lectures or homework problems. But the price for a non-trivial, representative operating systems project in terms of student efforts and computing costs is formidable. What is being debated is whether the experience gained from such a project is worth the price.

During the academic year 1974/75, an undergraduate operating systems course based on an implementation project was developed, carefully studied, and adopted at Rutgers University. The juniors and seniors taking this course have considerable applications programming expertise, but have had little exposure to systems programming and system organization. Under these circumstances, it appeared futile to attempt to convey operating systems principles without first providing an intuitive understanding of the

overall operation of a computer system. But we were also painfully aware of the price which would have to be paid to achieve this goal. Clearly, this project could only be successful if the price could be kept affordable and if the benefit to students was outstanding. We settled on a number of necessary criteria for keeping the price/benefit ratio low. (1) The project should involve an entire, though simple, system to illustrate both the overall operation and the implementation details. (2) Computing costs should be kept reasonable, although it was clear from the beginning that they would be high. (3) The system should be representative of contemporary ones, but it should not be based on any particular brand. This would allow the use of advanced effort-saving architectures and also simplify the introduction of implementation-independent concepts. (4) The opportunity should be used to introduce students to the intrinsic management problems of large software projects which are typically absent in academic courses.

From the first criterion we concluded that the project should involve both a hardware simulator and an operating system. This decision, which resulted in an integrated design methodology for both hardware and software, allowed us to design a machine which is particularly suited for the implementation of operating systems. It is termed COS for Computer for Operating Systems and exists in two versions [9]. COS-1 consists only of a processor and primary memory; it serves to get the project started fast. The downward compatible COS-2 includes the periphery, but its processor is also enhanced with hardware features (instructions, interrupt system, registers) which support the operating system. In view of the increasing availability of microcode, it is felt that the integrated design methodology is also very promising for commercial systems. In any event, it cut the effort for implementing the operating system in half without appreciably complicating the hardware simulator for COS-2. To cut computing costs, it was decided that the system should not process symbolic information; we estimated that symbolic programs would require about 20 times more storage than their object code equivalents. Instead, a crossassembler was developed and made available to students in order to ease their workload. The third criterion resulted in an operating system whose mode of operation is similar to the "THE"-Multiprogramming System [10]. This system is written in assembly

language, but the assembly language has the appearance of an ALGOL-derivative in order to relate to the notation of most textbooks on operating systems. The system is named COSMOS, an abbreviation for COS Multiprogramming Operating System [11]. COSMOS spools user jobs from a card-reader to a drum. Whenever there is sufficient primary memory available, a job is loaded for execution. Operating systems calls are available for I/O operations and termination. After the latter, a job's output is printed on the lineprinter with performance statistics. In satisfaction of the fourth criterion, COSMOS is structured in levels of abstraction and the entire project is assigned incrementally. Students work in teams of two and often experience for the first time that they have to live with code which they wrote two months earlier.

The project was successfully completed by most students who took the course during the last two years. Except for the crossassembler, the entire system is implemented by a team of two students, but detailed design specifications in terms of reference manuals and flowcharts are handed out. The project is structured in the form of six assignments, each of which build on all previous ones:

1. Simulation and testing of COS-1.
2. Simulation and testing of COS-2.
3. Run of a stand-alone periphery test program.
4. Testing of the COSMOS spooling system.
5. Run of a single user job under COSMOS.
6. Evaluation of runs of batches of user jobs.

The first three assignments represent the hardware simulator and are completed after about one month. The last three assignments representing the operating system require another month for completion. During the last third of the semester, advanced concepts are presented as solutions to the inadequacies of the implemented system. Due to the expertise gained from the implementation, students tend to appreciate the value of more abstract principles which can thus be dealt with at a rapid pace. It is this third phase of the course which indicates that the benefits derived from the project more than justify its price. The feedback and the enthusiasm of students corroborate this indication.

THE HARDWARE AND ITS SIMULATOR

The configuration of the simulated COS-2 hardware complex has been designed to support the operation of the COSMOS system. It is illustrated in Figure 1. Only essential devices have been included, but both modes of direct I/O (cardreader and lineprinter) and indirect I/O (Drum controller) are represented. Primary memory assumes the central position for data transfers, while the CPU (and its console) serves as the system's control center. While this configuration is representative of contemporary minicomputer systems, its restriction to essential devices also allows for the implementation of a detailed simulator within a reasonable period of time.

The simulator consists of four major modules: a subroutine for the instruction execution cycle, a main driver which corresponds to the console and repeatedly calls the execution cycle subroutine, a common module containing the storage declarations and constants like masks and fields, and another subroutine simulating the combined cardreader and crossassembler (see below). The drum system is simulated by the instruction cycle subroutine which counts real time in terms of the number of instructions executed. The simulator may be written in any high-level language. In the past two years, FORTRAN was chosen for reasons specific to our environment, but also because this lowest of all high-level languages quite appropriately reflects the nature of hardware operations. In fact, many of the FORTRAN statements may be interpreted as microinstructions, thus lending themselves to the introduction of firmware. The final versions of the COS-2 simulator without the cardreader/cross-assembler subroutine (which was written in PL/I) consisted of about 500 FORTRAN STATEMENTS.

CPU AND PRIMARY MEMORY

The CPU contains many of the standard features for its task of instruction execution and system supervision. Four registers, program counter, instruction register, accumulator, and index register, serve the obvious purposes. For simplicity, a fixed, single-word instruction format has been adopted. In addition to the operation code and four addressing modes (indexing and indirection in any combination), an instruction contains two address fields. The addition of a second address field to the originally contemplated single-address format (a seemingly unimportant detail) had a considerable impact on both code compaction (about 30%) and code legibility. The latter is due to the fact that, in assembly language, many quantities can be referred to be meaningful symbolic names, rather than by the semantically irrelevant names of the registers in which they reside. The operating system is aided in its task by four more CPU registers: interrupt and trap register, arming register, mode register, and state pointer register. The latter contains a pointer to the process state record (PSR), the data structure describing the state of the currently running process (Figure 2). Interrupts and traps are serviced by a multi-level priority interrupt system which saves the entire state of the interrupted process and loads the initial state of the interrupt service routine. This control transition is performed completely in hardware and makes use of both the state pointer register and an interrupt transfer vector in standard low core locations.

Virtual memory is supported by a paging map. The virtual address space covers 1K words consisting of 8 pages of 128 words each. Words, both primary memory locations and registers, are 36 bits long; they have been dimensioned after the integer size of the IBM 370 system on which the simulator is developed and run. Primary memory consists of 2K words or 16 pages of 128 words each. It is accessed from the CPU via the paging map. Page 0 contains a number of standard locations ("low core") for communications among the various devices. These standard locations are used for bootstrapping, timers, the drum queue pointer, and the interrupt transfer vector.

Figure 1. COS/2 system configuration.

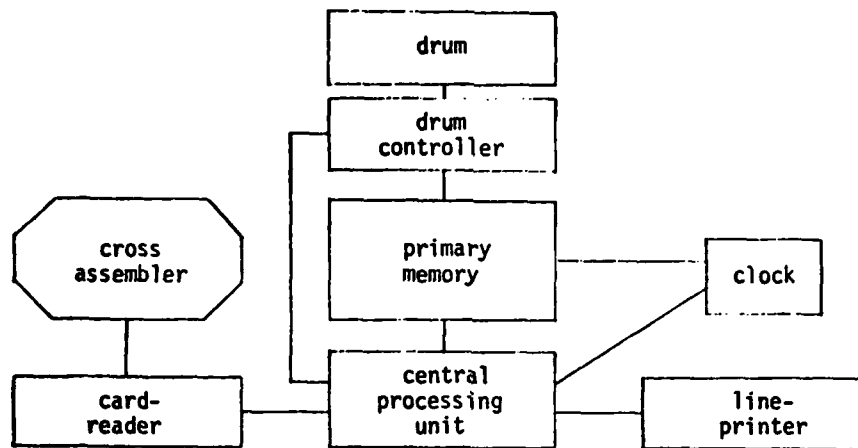


Figure 2. The format of a process state record (PSR).

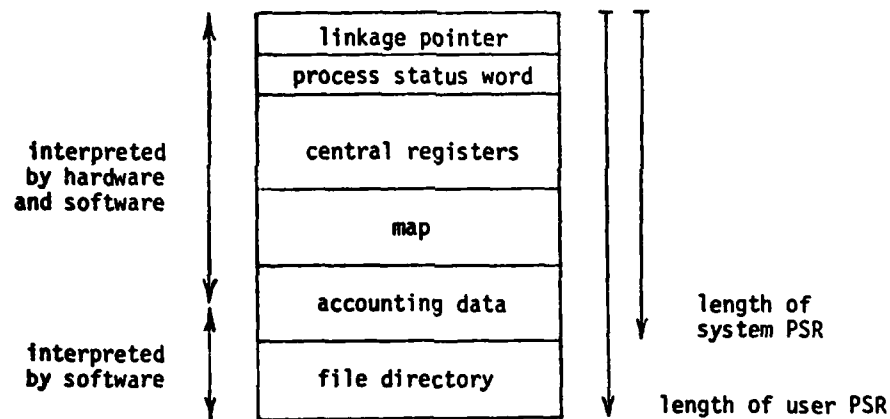
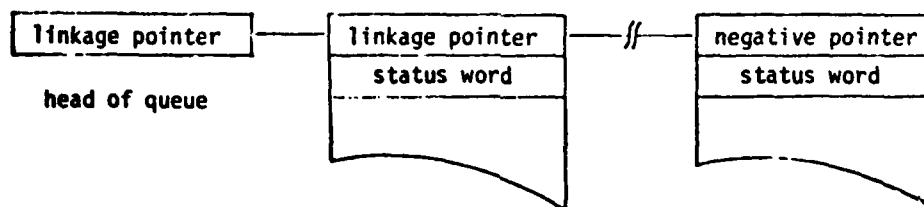


Figure 3. The standard format for COSMOS queues.



The role of the console has been stressed for pedagogical reasons. First, its switches and indicators allow for a completely manual operation of the entire hardware complex. Aided by a breakpoint facility, a student can thus slow down the operations of the system and inspect its state via the indicators (printed out by the main driver). In this "hands-on" mode of operation, the student can debug his/her programs (including the simulator, i.e. his/her hardware configuration) and gain familiarity with the system in a manner comparable to a systems programmer or a programmer/operator of the 1950's. On the other hand, no real hardware must be provided, thus avoiding scheduling problems as well as hardware failures. Second, the bootstrap mechanism which loads stand-alone programs (including the operating system) from the cardreader into primary memory can be activated from the console. Since most students' knowledge about operating systems is restricted to external characteristics like command language and operating systems calls, the concept of how a system is brought up often challenges their imagination. We found that the simulation not only reduced the effort for conveying this concept, but also illustrated the functional equivalence of hardware and systems software. This equivalence is an important notion, since it lends itself to the introduction of microcode. Microcode, in turn, justifies the implementations of a number of operating system functions as instructions. The inclusion of these functions in the instruction set resulted in a considerable simplification of the COSMOS code and, thus, aided in keeping the programming effort within reasonable bounds.

In addition to a fairly standard complement of about 40 instructions for data movements, arithmetic and logic operations, branching, and input/output, the instruction set contains 13 operating systems support instructions. The criteria for their inclusion were high execution frequency and/or potential for code compaction. Neither the instruction set nor the instruction format were frozen until a version of COSMOS had been coded in a hypothetical open-ended language which eventually converged to the assembly language. This approach allowed for a frequency analysis of instructions and resulted in the deletion of superfluous ones as well as in the adoption of the operating systems support instructions.

According to Werkheiser's classification [12], miniprimitives (equivalent in power to conventional machine instructions) and midprimitives (equivalent to the system macro) evolved. The former were designed to support the manipulation of the COSMOS data structures, in particular bit-tables and queues. Two instructions inspect and conditionally update bit-tables and indicate their action upon return. They are used for the allocation of drum and core pages. A standard format for queues was adopted which includes both linkage pointers and status words (Figure 3). Operations to get, return, append, insert, and find a record are supported. They are used primarily to manipulate process state records and drum transfer records (DTR's, describing requests for drum I/O). Midprimitives are included to support level crossings. User processes enter system processes via an operating

system call instruction (OS CALL) which causes a trap. Traps as well as interrupts save the state of the current process in its PSR and load the state of the next process from its PSR. An identical operation is performed by the instruction SWITCH PROCESS, but it obtains a pointer to next PSR from a register rather than low core. Since this operation provides the only means of setting the paging map, it constitutes a central part of the protection mechanism in COSMOS.

Peripheral Devices

The complexity of the peripheral devices has been reduced to a minimum. The cardreader contains a hopper which is filled with the decks of a number of jobs. Each deck consists of a JOB card, followed by program cards, a DATA card and input data cards (optional), and an END card. The information on a card is encoded in 32 bits and corresponds to a machine word. Upon execution of the instruction READ CARD the contents of the next card in the hopper are transferred to the accumulator and the instruction skips a variable number of times depending on the type of card (or lack thereof). The cardreader also serves as the input device for bootstrapping.

The lineprinter can be activated via two instructions. PRINT LINE prints the contents of the accumulator as an eight-digit hexadecimal integer after the printout paper has been advanced one line. Thus, printout will typically consist of a single column of integers. A special purpose instruction, HEADER, is provided to print the header preceding the output of a user job. HEADER prints several lines of symbolic and numerical information which is extracted from a process state record. The information contains job statistics and possibly indications of an abnormal termination.

The drum system represents an indirect I/O device. The storage capacity is 16 pages of 128 words. Transfers are on a page basis and are controlled by the drum controller which is started from the CPU by executing a START DRUM instruction. The controller obtains the transfer parameters from a drum transfer record pointed to by a standard location in low core. The controller informs the CPU about the completion of a transfer by issuing an interrupt.

THE COSMOS OPERATING SYSTEM

Most contemporary definitions identify operating systems as programs, i.e. software. We feel that these definitions are too narrow, that they stress an implementation aspect rather than concentrating on functional aspects. We prefer to think of an operating system as a system of components which operates a computer system in a user-oriented and efficient manner. Whether certain functional components are represented by hardware, microcode, software, or even a human operator, will be dictated by the current state of technology. An alternative to the traditional software overlay techniques has long been offered in hardware supported virtual memories, and in trillion-bit memories mechanical devices now cost-

effectively perform the traditionally human task of mounting removable storage media. In view of the seemingly unsurmountable difficulties encountered in large software systems, it appears worthwhile to reconsider the assignment of operating systems functions to specific implementation media.

Compared to commercial systems, our educational COSMOS system may well be characterized as a toy system. But there is at least one thing which they have in common: the effort required for their implementations approaches the limit of the available manpower. OS/360 devoured 5000 man-years for its design, construction, and documentation [13]. This must be compared to a couple of man-months which can be put in by a team of two students registered for a normal load of four courses during one semester. Clearly, this analogy must not be pushed too far. But it vividly demonstrates the effect of measures which result in cutting the required effort in half. Since the implementation project includes both a hardware simulator and COSMOS code, one important design parameter concerned the distribution of functional capabilities over hardware and software. We believe that the successful completion of this project within one semester by almost all students is a direct consequence of our integrated design methodology and the resulting simplification of the overall system.

COSMOS is coded in assembly language and its loadmodule fits into a single virtual address space (1K words). The actual code, as opposed to data structures and buffers, consists of less than 500 instructions.

Mode of Operation

In many respects, the mode of operation in COSMOS is similar to Dijkstra's "THE"-Multi-programming System. Unnecessary sophistication has been sacrificed in order to keep the implementation effort manageable. For instance, user jobs cannot leave information behind them in the system after termination. For the same reason, no symbolic information can be processed; the system relies on a crossassembler, which is coupled with the cardreader, for symbol manipulation. Furthermore, only one language processor (for assembly language) is available, and a standard command sequence is implied (rather than providing a more versatile command processor).

COSMOS is brought up by bootstrapping from the cardreader. After initialization, it enters the stationary mode in which it accepts user jobs from the cardreader for processing. A job must be submitted in the following order: JOB card, program, DATA card and input data (optional), and END card. Syntactically incorrect jobs are flushed by the crossassembler and are not passed on to the cardreader. At any one time, several jobs may be in the system at various stages of completion. Any one job, however, will be processed in the following sequence. After being spooled to the drum, a job's program is loaded into core and then given control of the CPU. For input, the program may execute an OS CALL which will provide it with the next input word from the input data page on the drum. Another OS CALL causes the transfer of an output word to

the output data page on the drum. After termination of a user job, its output is printed on the lineprinter together with some statistics. This default sequence renders an explicit command processor unnecessary; its function is performed by the input spooling system. After spooling in the last user job, COSMOS will continue to process the resident jobs and halt after the last one has terminated.

Crossassembler

COSMOS is written in the assembly language of COS [14]. Thus, there is a one-to-one correspondence between assembly language statements (except pseudo operations) and machine instructions which have been implemented in detail in the simulator. On the other hand, the language has been given the flavor of an ALGOL-like system programming language by using meaningful mnemonics and by enclosing the address fields in parentheses, much like parameters in a subroutine call. See Figure 5 for a sample program. The syntax of the assembly language has been adopted for purely pedagogical reasons, since many textbooks use some sort of ALGOL-derivative for the illustration of algorithms. During the first year, when the cross-assembler accepted the standard three-letter mnemonics, many students found it difficult to relate textbook examples to their own machine language code. The simple change in the syntax apparently resolved that problem during the last semester.

If programs were entered via the cardreader in symbolic form, they would require about 20 times more storage space than machine code. To keep the cost of a simulation run down, it was decided to enter only binary information via the cardreader. This is accomplished by providing a crossassembler whose output is coupled with the cardreader. Students are provided with a loadmodule of the crossassembler which is written in PL/I. It is written as a subroutine which, when called, returns the next word of the previously assembled job as well as an indication about the nature of the corresponding card (instruction, control card, etc.). If no more object code words are available, crossassembler reads in the next job (symbolic program, input data, and control cards), assembles it, and returns the first word, the JOB card. To the hardware simulator, the cross-assembler subroutine appears to represent the cardreader, since it returns the equivalent of one card upon each call. In a simulation run, the input data to the cardreader/crossassembler subroutine consists of the COSMOS job (which will be bootstrapped) followed by an arbitrary number of user jobs.

COSMOS Structure

COSMOS is structured in levels of abstraction which are depicted in Figure 4. The assignment of functions to particular levels evolved from a number of considerations. The mode of operation determined the high-level functions of spooling, I/O, loading, termination, and printing. The concept of multiprogramming is implemented by the SCHEDULER one level below so that the high-level functions can be scheduled like a user job. The interrupt routines in level 4 provide real-time

dependence for all higher levels. Drum-multi-programming, realized by manipulating drum transfer records in the drum queue, is real-time dependent and thus one level below. the notion of a process may be defined abstractly, but in any implementation it is simply the quantity described by the corresponding operating system data structure, e.g. the PSR. In manipulating these data structures, level 2 provides the process notion. Finally, data structures must be allocated before they can be manipulated. Thus, storage allocation has been assigned to the lowest level in the hierarchy.

The assignment of functions of the hierarchy in Figure 4 evolved gradually during the course of several top-down and bottom-up design iterations. This design methodology also involved decisions about the implementation media for the various modules. All modules in levels 1 and 2 are implemented as instructions, thus considerably simplifying and compacting the COSMOS code. Semaphore operations were not included since they are not representative of commercial systems which are graduates are most likely to deal with. Instead, they are introduced at the end of the semester as elegant mechanisms for handling synchronization problems. The latter are emphasized by level 3 which is implemented as a shared (and interruptible) subroutine. All modules above level 3 are implemented as processes, i.e. programs whose execution state is described by a PSR.

The SWITCH PROCESS instruction passes control among processes. A similar operation is performed by the interrupt system for passing control to processes in level 4. These interrupt processes, however, use the standard SWITCH PROCESS instruction to return to the interrupted process. Interrupt processes may be interrupted by an interrupt of higher priority. The SCHEDULER in level 5 initially obtains control from the bootstrap mechanism. Thereafter, the control transfer to a schedulable process in level 6 and the return to the SCHEDULER is accomplished by SWITCH PROCESS INSTRUCTIONS. With respect to scheduling, we followed the basic guideline that mechanisms should be implemented in hardware while policy aspects are more flexible in a software implementation [15, 16]. The task of activating, deactivating, blocking, and unblocking processes, accomplished by setting and/or resetting bits in the status words of the corresponding PSR's, is distributed among the COSMOS processes. For example, since loading of a job depends on the successful completion of the spooling operation, the spooling process which finishes this operation will activate the loader on behalf of this job.

Processes and Their Functions

After system initialization, the SCHEDULER loops scanning the process state record queue and picking the ready (i.e. active, but not blocked) job with the highest priority for execution. The search is performed by the FIND RECORD instruction which compares the second parameter under a mask in the accumulator with the status words in the queue (Figure 5). FIND RECORD skips if a match is found and returns a pointer to the matching PSR in the index register. SWITCH PROCESS transfers control to the process specified by the

pointer in the index register; its address field denotes the address at which the process relinquishing control will start execution at its next activation. If no processes are ready, the SCHEDULER checks whether there are any blocked ones; if not, the simulation run is terminated by executing a HALT.

The Drum Interrupt Process cleans up after a transfer and restarts the drum if more requests are pending. It also supports the FILE SYSTEM in performing user I/O. The Trap Process handles OS CALL's much like OS/360 as well as protection violations. User programs exceeding their time limits (set by the SCHEDULER) lose control to the Timer Interrupt Process when the timer goes off.

A double-buffering scheme with dynamic buffer allocation is used for spooling jobs from the cardreader to the drum. The two processes RCARD and WCARD read a page into core and write it to the drum respectively. They synchronize each other and also initialize a PSR for the arriving job. After a job has been spooled in, the LOADER will transfer its code to primary memory as soon as there are enough pages available. A simple static pre-loading scheme is used. User jobs read input data and write output data sequentially by executing the appropriate OS CALL's which in turn cause an activation of the FILE SYSTEM. Buffers required for the input or output page are allocated dynamically and their locations as well as the access indices are maintained in the PSR of the requesting user. The end of execution of a user job is signaled to COSMOS by another OS CALL. The TERMINATOR WILL subsequently release all allocated resources except for the output data page. Jobs may also terminate abnormally by executing an illegal instruction, by causing a memory violation, or by exceeding their quanta. A user job's output is preceded by statistics and printed by the PRINT process which finally returns the job's PSR to the freelist.

Conclusion

The COSMOS project has proven itself as a viable teaching tool. Its scope is modest, yet it is representative of contemporary systems, and its structured and integrated design utilizes advanced concepts. While the implementation of the hardware simulator provides insight into the underlying mechanisms, the coding of the operating system stresses the interrelationship of these mechanisms in forming an operational computer system. The performance of this system can be evaluated as a function of hardware and software parameters by running batches of user jobs. Thus, the project provides a framework for the study of both design and system behavior.

Despite its numerous benefits, the project could not have succeeded without the extensive design and documentation efforts which focussed on a reduction of the implementation effort without trivializing the system. The applied integrated design methodology, which resulted in an enhancement of the hardware for operating systems support, appears to be applicable to real systems. At a time when it is not uncommon that 50 percent of the cycles of a "general-purpose" system are consumed by operating systems code, it may well be justified to replace the general-purpose processor

Figure 4. Levels of abstraction in COSMOS.

LEVEL	ABSTRACTIONS	FUNCTIONS	MODULES	DATA STR.
7		user problem execution		
6	virtual user memory, files, OS CALL's	user support (spooling, I/O, loading, termination, printing)	RCARD, WCARD, LOADER, FILE SYSTEM, TERMINATOR, PRINT	
5	CPU multi-programming	CPU allocation	SCHEDULER	
4	real-time independence	service of interrupts and traps	Timer Interrupt Process, Drum Interrupts Process, Trap Process	
3	drum multi-programming	allocation of drum controller	PUTDRQ	
2	CPU and drum processes	accesses of PSR's and DTR's	Queue instructions, SWITCH PROCESS, START DRUM	PSR queue, drum queue
1	storage allocation	accesses of allocation words	ALLOCATE, DEALLOCATE	core and drum allocation words

Figure 5. Assembly language code for the COSMOS scheduler.

```

* SCHEDULER
*****
*
* INITIALIZATION (GETS CONTROL FROM BOOTSTRAP LOADER)
*
S$INIT:  XR LOAD      (LC$SCH);
         SP=XR;
         IT RESET    (NEGONE);
         AR RESET    (NEGONE);
         AR SET      (INTBITS);
         AC LOAD     (ABIT);
         INITIALIZE SP
         CLEAR IT
         CLEAR AR
         ENABLE AND ARM LEVELS
         INIT PATTERN FOR MATCHING

* SCHEDULING
*
S$ENTRY:  MOVE        (LC$RTC, S$QUANT);
         FIND RECORD  (PSRQ, ABBITS);
         GO TO        (S$ACHECK);
         SWITCH PROCESS (S$ENTRY);
         SET TIMER
         FIND READY PROCESS
         NO PROCESSES READY
         PASS CONTROL TO READY PROC

* CHECK FOR BATCH COMPLETION
*
S$ACHECK:  FIND RECORD  (PSRQ, ABIT);
         HALT;
         GOTO          (S$ENTRY);
         FIND AN ACTIVE PROCESS
         NONE, BATCH COMPLETED
         OBVIOUSLY BLOCKED, KEEP
         SCANNING PSRQ WHILE WAITING FOR ANY
         PROCESS TO BECOME READY
*****
* LOCAL VARIABLES
*
S$QUANT:  CONSTANT    ('FFFFFFF);
         LARGE NEGATIVE QUANTUM

```

by a special-purpose operating systems processor. Microcode technology is ready for this application.

Students have responded quite positively to the adoption of the project. Several extensions have been designed and partially implemented in independent study projects. A further, interactive derivative has been written in SIMULA on a PDP-10 and currently serves as the framework for graduate research in the area of protection.

References

1. ACM Curriculum Committee on Computer Science. "Curriculum 68," Comm. ACM, 11, 3, March 1968.
2. Cosine Committee of the Commission on Education of the National Academy of Engineering, "An Undergraduate Course on Operating Systems Principles," Washington, D.C., June 1971.
3. Madnick, S. E. and Donovan, J. J. Operating Systems. McGraw-Hill, New York, 1974.
4. Shaw, A. C. The Logical Design of Operating Systems. Prentice-Hall, Englewood Cliffs, New Jersey, 1974.
5. Tsichritzis, D. C. and Bernstein, P. A. Operating Systems. Academic Press, New York, 1974.
6. Habermann, A. N. Introduction to Operating System Design. Science Research Assoc., Chicago, Illinois, 1976.
7. Lamie, E. L. "Using GPSS to Teach Operating Systems Concepts," Proc. ACM Tech. Symp. Computer Science and Education, Anaheim, California, February 1976.
8. Hughes, C. E. and Pfleeger, C. P. "Assist-V-A Tool for Studying the Implementation of Operating Systems," Proc. ACM Tech. Symp. Computer Science and Education, Anaheim, California, February 1976.
9. Ruschitzka, M. "COS - Model 1 Reference Manual," "COS - Model 2 Reference Manual," CS 316 Class Notes, Dept. of Computer Science, Rutgers University, New Brunswick, New Jersey, 1976.
10. Dijkstra, E. E. "The Structure of the "The"-Multiprogramming System," Comm. ACM, 11, 5, 1968.
11. Ruschitzka, M. "COSMOS Reference Manual," CS 316 Class Notes, Dept. of Computer Science, Rutgers University, New Brunswick, New Jersey, 1976.
12. Werkheiser, A. H. "Microprogrammed Operating Systems," Proc. 3rd Annual Workshop on Microprogramming, October 1970.
13. Brooks, F. P., Jr. The Mythical Man-Month. Addison-Wesley, Reading, Massachusetts, 1975.
14. Ruschitzka, M. "CAL-CAROL Reference Manual," CS 316 Class Notes, Dept. of Computer Science Rutgers University, New Brunswick, New

Jersey, 1976.

15. Lampson, B. W. "A Scheduling Philosophy for Multiprocessing Systems," Comm. ACM., 11, 5, 1968.
16. Levin, R. et al. "Policy/Mechanism Separation in HYDRA," Proc. Fifth Symp. on Operating Systems Principles, University of Texas, Austin, November 1975.

The project reported in this paper was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under grant DAHCIS-73-G6. Author's address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903

**Best
Available
Copy**

COS - Model 1
REFERENCE MANUAL

1 INTRODUCTION

COS-1 is the basic model of the COS (Computers for Operating Systems) series. It consists of a central processing unit (CPU) and a 2K-module of 32-bit primary memory (PM) which is addressed from 0 to 2047 (larger addresses are cut back to 11 bits and location 0 is inaccessible). COS-1 is a paging system and features direct, indirect, and indexed addressing modes. The pagesize is 128 words (7-bit displacement) and the virtual address space contains eight pages. Thus, virtual addresses are ten bits long.

Arithmetic is performed in two's complement form on 32-bit integers. Besides arithmetic operations, the instruction repertoire includes logical, branching, and register operations as well as a HALT instruction.

2 ARCHITECTURE

The four major functions of the CPU are reflected in the system architecture. Instruction execution is controlled by the control unit (CU). Arithmetic and logical operations are performed in the arithmetic and logical unit (ALU). The local storage unit (LSU) contains the relocation map and other registers. The system may be operated manually via the console. Figure 1 illustrates.

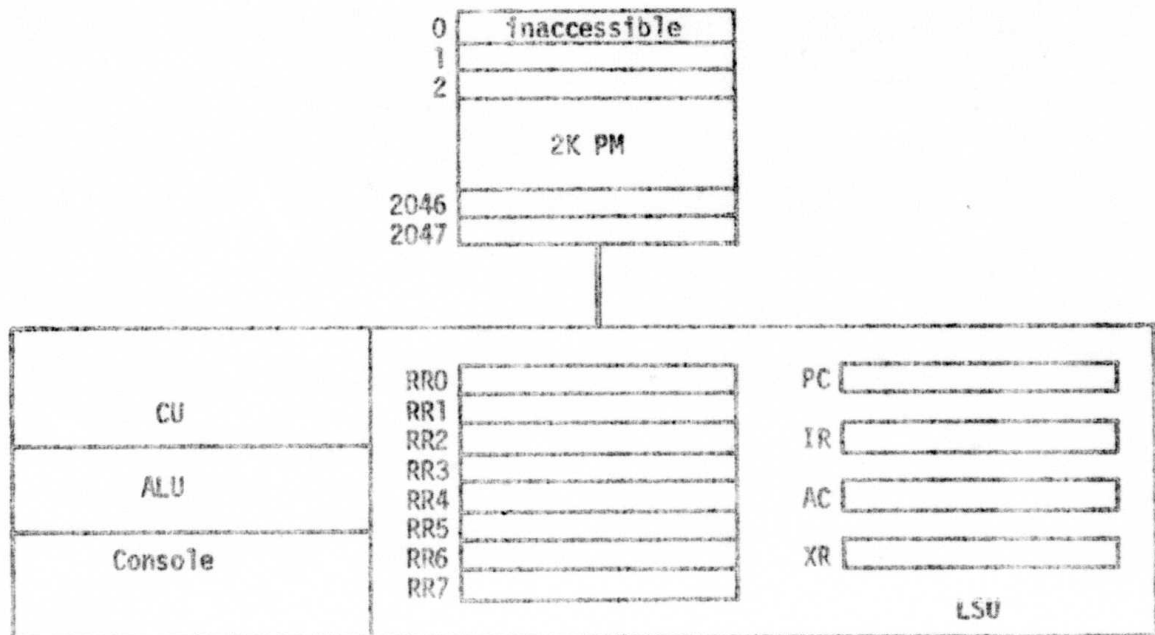


Figure 1: The COS-1 architecture.

The 32-bit registers in the LSU serve the following functions:

The program counter (PC) contains the virtual address of the current (or the next) instruction. Only the rightmost ten bits are significant.

The instruction register (IR) contains the currently executed instruction.

The accumulator (AC) provides temporary storage for intermediate operands.

The index register (XR) is used primarily for address computations (indexing). It may also be used for temporary storage.

The eight relocation registers (RR0 through RR7), or map, provide the current mapping of virtual into physical page numbers. Figure 2 illustrates their role in the mapping process. With 2K of PM, the least significant four bits suffice to specify any physical page number. All other bits are ignored, except for bit 0 which - when set - indicates an invalid entry. Accesses via invalid entries are treated as memory violations.

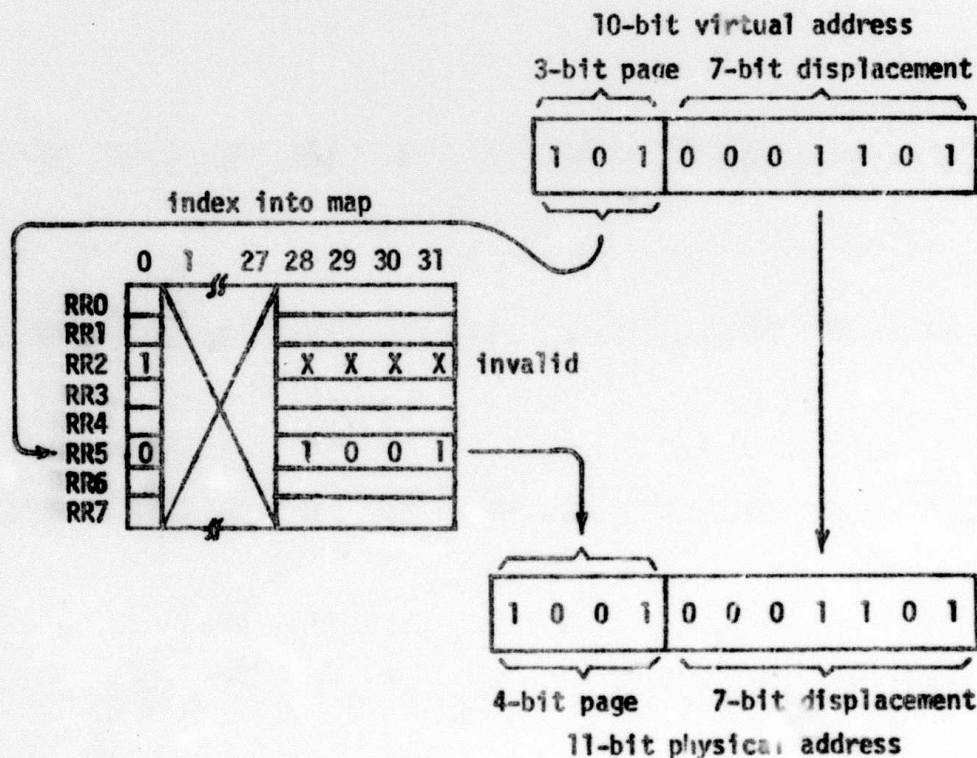


Figure 2: Mapping of a virtual address.

3 INSTRUCTION FORMAT

A double address instruction format has been adopted for the COS series. However, of the two addresses embedded in the instruction, only address A2 may be modified via indirection and/or indexing (Figure 3). For address A1, direct addressing mode must be used.

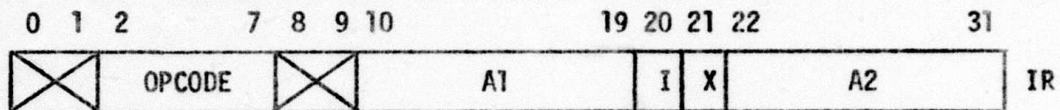


Figure 3: COS instruction format.

An instruction contains four different fields:

The operation code (OPCODE) field is six bits long. The two leftmost bits specify one of four instruction types.

The two address fields (A1 and A2) span ten bits each. Thus, every virtual location can be addressed directly.

The addressing mode field consists of the index bit (X) and the indirect bit (I). Thus, there are four different addressing modes with respect to address A2:

<u>I X</u>	<u>A2 addressing mode</u>
0 0	direct
0 1	indexed
1 0	indirect (one level only)
1 1	indirect before indexed (one level of indirection)

Figure 4 sketches the algorithm for the generation of the effective virtual addresses V1 and V2. MAP(V2) denotes the physical address of the indirect location. Since some instructions do not make use of both addresses, the generation of the effective physical addresses L1 and L2 is requested only when needed, i.e. after the processor dispatched on the operation code.

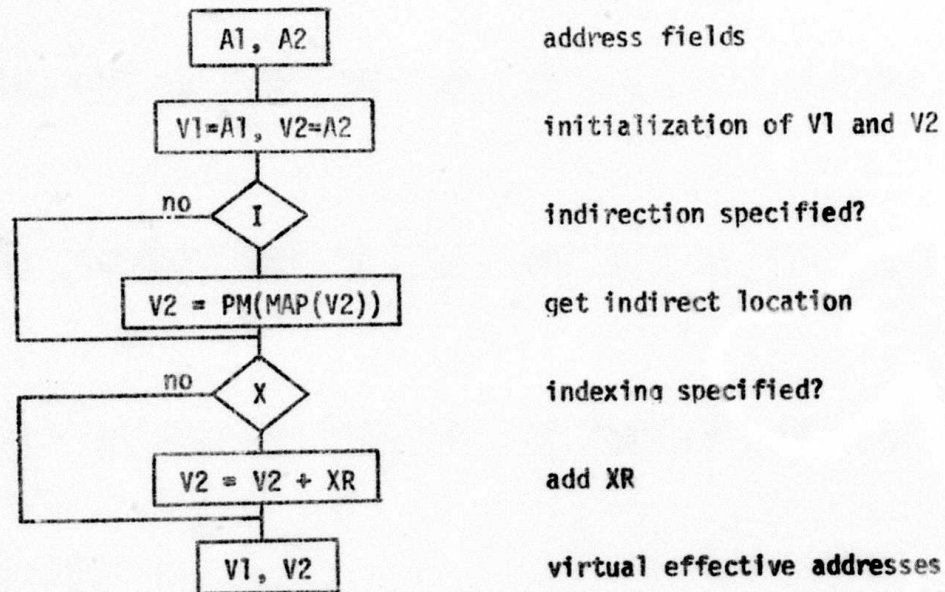


Figure 4: Algorithm for the virtual effective address generation.

4 INSTRUCTION REPERTOIRE

In the COS series, instructions are grouped according to four types. The type of an instruction is encoded in the leftmost two bits of its operation code:

<u>leftmost bits</u>	<u>type</u>
0 0	0: register and transfer
0 1	1: arithmetic, logical and branching
1 0	2: input-output
1 1	3: operating systems support

In the following functional description, L1 and L2 represent the physical effective addresses obtained from A1 and A2 via the effective virtual address generation and a subsequent mapping. Except for the HALT instruction (type 2), the COS-1 instruction set contains only instructions of the types 0 and 1.

OPCODE		MNEMONIC	ADDR. USED	OPERATION
HEX	DEC			

Type 0: Register and transfer

00	0	unused		
01	1	AC CLEAR		AC=0
02	2	XR CLEAR		XR=0
03	3	AC INCREMENT		AC=AC+1
04	4	XR INCREMENT		XR=XR+1
05	5	XR=AC		XR=AC
06	6	AC=XR		AC=XR
07	7	AC LOAD	A2	AC=PM(L2)
08	8	AC STORE	A2	PM(L2)=AC
09	9	XR LOAD	A2	XR=PM(L2)
0A	10	XR STORE	A2	PM(L2)=XR
0B	11	MOVE	A2,A1	PM(L2)=PM(L1)
0C	12	BACK MOVE	A2,A1	PM(L1)=PM(L2)
0D	13	unused		
0E	14	unused		
0F	15	unused		

Type 1: Arithmetic, logical and branching

10	16	ADD	A2,A1	PM(L2)=PM(L2)+PM(L1)
11	17	SUBTRACT	A2,A1	PM(L2)=PM(L2)-PM(L1)
12	18	MULTIPLY	A2,A1	PM(L2)=PM(L2)*PM(L1)
13	19	DIVIDE	A2,A1	PM(L2)=PM(L2)/PM(L1)
14	20	AND	A2,A1	PM(L2)=PM(L2)APM(L1)
15	21	OR	A2,A1	PM(L2)=PM(L2)VPM(L1)

OPCODE HEX DEC		MNEMONIC	ADDR. USED	OPERATION
16	22	COMPLEMENT	A2,A1	$PM(L2) = \overline{PM(L1)} = -PM(L1) - 1$
17	23	unused		
18	24	GO TO	A2	$PC = V2$
19	25	IF >0 GO TO	A2,A1	if $PM(L2) > 0$: $PC = V1$
1A	26	IF =0 GO TO	A2,A1	if $PM(L2) = 0$: $PC = V1$
1B	27	IF <0 GO TO	A2,A1	if $PM(L2) < 0$: $PC = V1$
1C	28	IF <=0 GO TO	A2,A1	if $PM(L2) \leq 0$: $PC = V1$
1D	29	IF >=0 GO TO	A2,A1	if $PM(L2) \geq 0$: $PC = V1$
1E	30	LOOP	A2,A1	$XR = XR + 1$; if $XR < PM(L2)$: $PC = V1$
1F	31	CALL	A2	$PM(L2) = PC$ (store return address); $PC = V2 + 1$

Type 2: Input-output

20	32	HALT		$HLT = 1$ (terminates execution)
21	33	unused		
2F	47			

Type 3: Operating systems support

30	48	unused		
3F	63			

RUTGERS UNIVERSITY - Department of Computer Science - CS 416/18
M. Ruschitzka
Spring 1977

COS - Model 2
REFERENCE MANUAL

1. INTRODUCTION

COS-2 is an extension of COS-1; it has been designed to support the implementation of operating systems. While COS-2 is fully downward compatible, allowing COS-1 programs to run without modifications, its processing power considerably exceeds that of COS-1. This is possible due to the following additional features:

- Peripheral devices (card reader, line printer, drum system, timers)
- Interrupt system
- System and user mode
- State pointer register
- Input-output and operating systems support instructions

This manual contains information about the additional features not incorporated in COS-1. For a description of the basic model see the reference manual for COS-Model 1.

2. ARCHITECTURE

The architecture of COS-1 has been enhanced by a card reader, a line printer, a drum system (consisting of a drum controller and a drum which provides storage for 16 pages or 2K words), and four CPU registers (Figure 1).

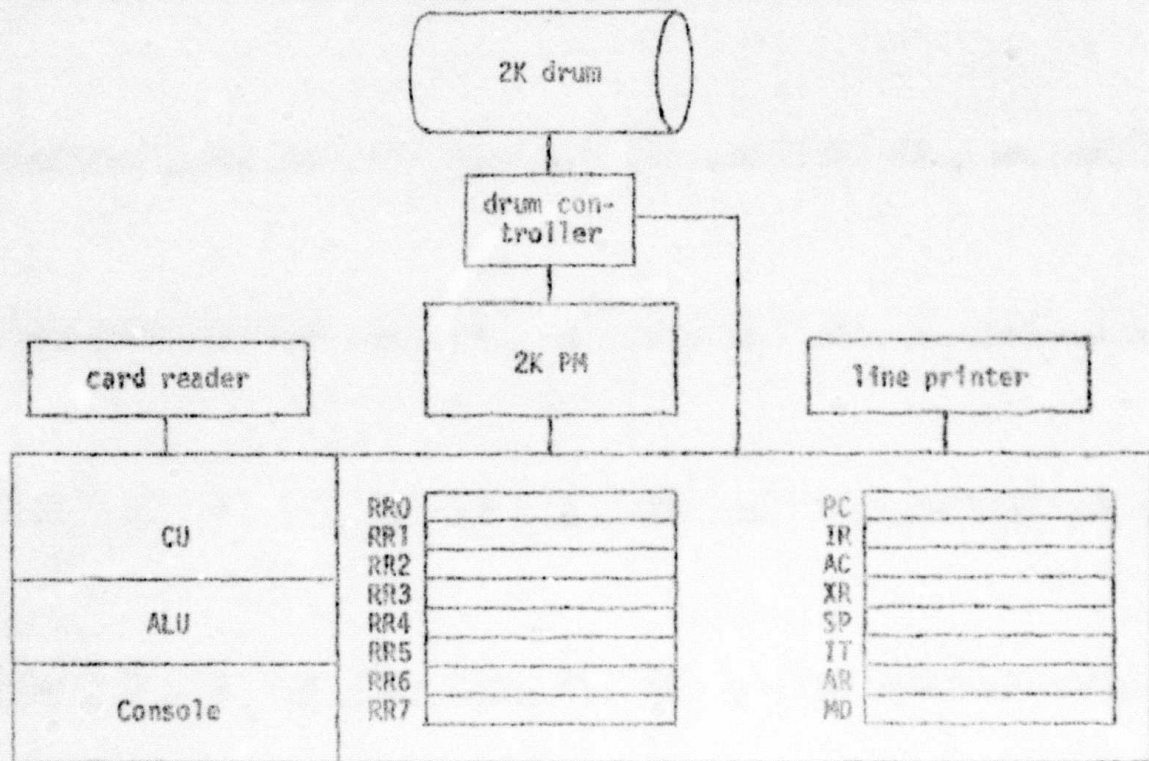
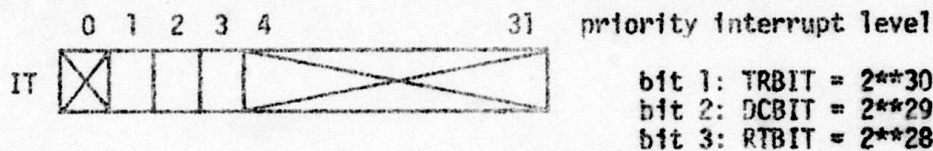


Figure 1: The COS-2 architecture.

The four additional registers serve the following functions:

The state pointer register (SP) contains a physical address pointing to the process state record of the currently executing process. Only the eleven least significant bits of this 32-bit register are relevant.

The bit positions in the interrupt and trap register (IT) represent priority interrupt levels. Bit 0 is not used. Priorities decrease from level 1 to level 31. Currently, only levels 1, 2, and 3 are assigned to traps, the drum controller, and the real time clock (timer) respectively. Bit positions 4 through 31 are therefore irrelevant. When a bit is set, it indicates that an event has occurred, but has not been serviced yet.



Interrupt levels must be armed in the arming register (AR) if the corresponding events are to be serviced. This can be done by selectively setting the appropriate bits. A level is disarmed by resetting its bit. The entire interrupt system can be disabled by resetting bit 0, the enable/disable bit. An interrupt will be serviced only if the system is enabled, its level is armed, and no interrupts of higher priority are pending.

The mode register (MD) serves two functions. Bit 0 indicates the mode in which the CPU is currently operating; if it is set, it indicates user mode, otherwise system mode. The remaining bits correspond to interrupt levels. If one of these bits is set, it means that the interrupt routine of the corresponding level is currently running. For example, when the drum interrupt routine (a system process) runs, only bit 2 will be set in MD.

3 CARD READER

The card reader contains a hopper which is filled with the decks of a number of jobs. Each deck consists of a JOB card, followed by program cards, a DATA card and input data cards (optional), and an END card. The information on a card is encoded in 32 bits and corresponds to one machine word. Special cards are used for control cards. The execution of a READ CARD instruction causes the next card in the hopper to be read into the accumulator AC. In addition, the card reader senses the type of a card (or the lack of one) and communicates it to a program by the number of skips which the READ CARD instruction performs:

JOB card	no skip
DATA card	one skip
END card	two skips
empty hopper	three skips
program or input data card	four skips

The card reader also serves as the input device via which standalone programs can be bootstrapped into the system.

4. LINE PRINTER

The line printer can be activated via two instructions. Upon execution of a PRINT LINE instruction, the contents of the accumulator AC are printed out as an eight-digit hexadecimal integer after the printout paper has been advanced one line. Thus, printout will typically consist of a single column of eight hexadecimal digits.

The HEADER instruction uses a more compact format. This instruction is meant to print the header preceding the output of a user job. Apart from the job number, the header contains job statistics (e.g. use of resources) and possibly indications of an abnormal termination.

5. DRUM SYSTEM

The drum system provides CS-2 with a secondary storage device. The storage capacity is 16 pages of 128 words each. Drum pages are addressed by their page number, i.e. from 0 through 15. Transfers between the drum and primary memory are on a page basis in the sense that each transferred block will consist of 128 words. But while a block must start on a page boundary on the drum, it may start at any location in primary memory. During the time of the transfer, the CPU may execute instructions in parallel.

The drum controller requires three parameters for the supervision of any particular transfer:

- drum address (a drum page number)
- primary memory address (a physical PM address)
- direction (read from drum or write onto drum)

These three parameters are passed to the drum controller in a record (a drum transfer record or DTR) which is pointed to by the contents of PM location 13 or D in hexadecimal. As illustrated in Figure 2, the direction is encoded as an integer; a nonpositive integer specifies reading from the drum, a positive one specifies writing. The first and the fifth words of a drum transfer record are for use by the operating system; they are ignored by the drum controller. The names for the pointer location and the displacements within a drum transfer record are also given in Figure 2.

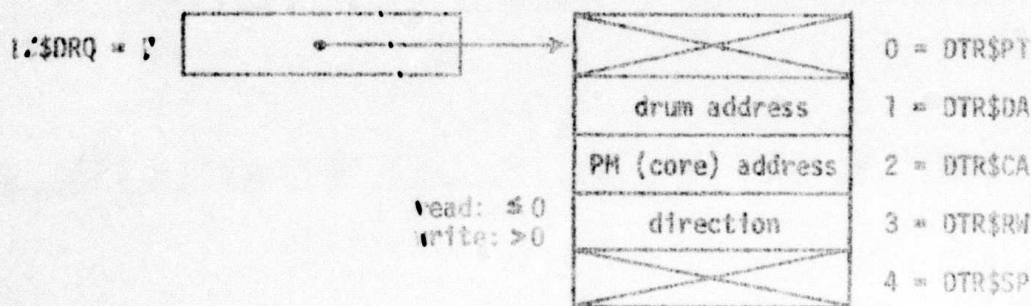


Figure 2: Pointer to and format of drum transfer record.

Since the drum controller has no access to the relocation registers, all primary memory addresses in Figure 2 are absolute physical addresses. This includes location D, the contents of location D, and the core address in the drum transfer record.

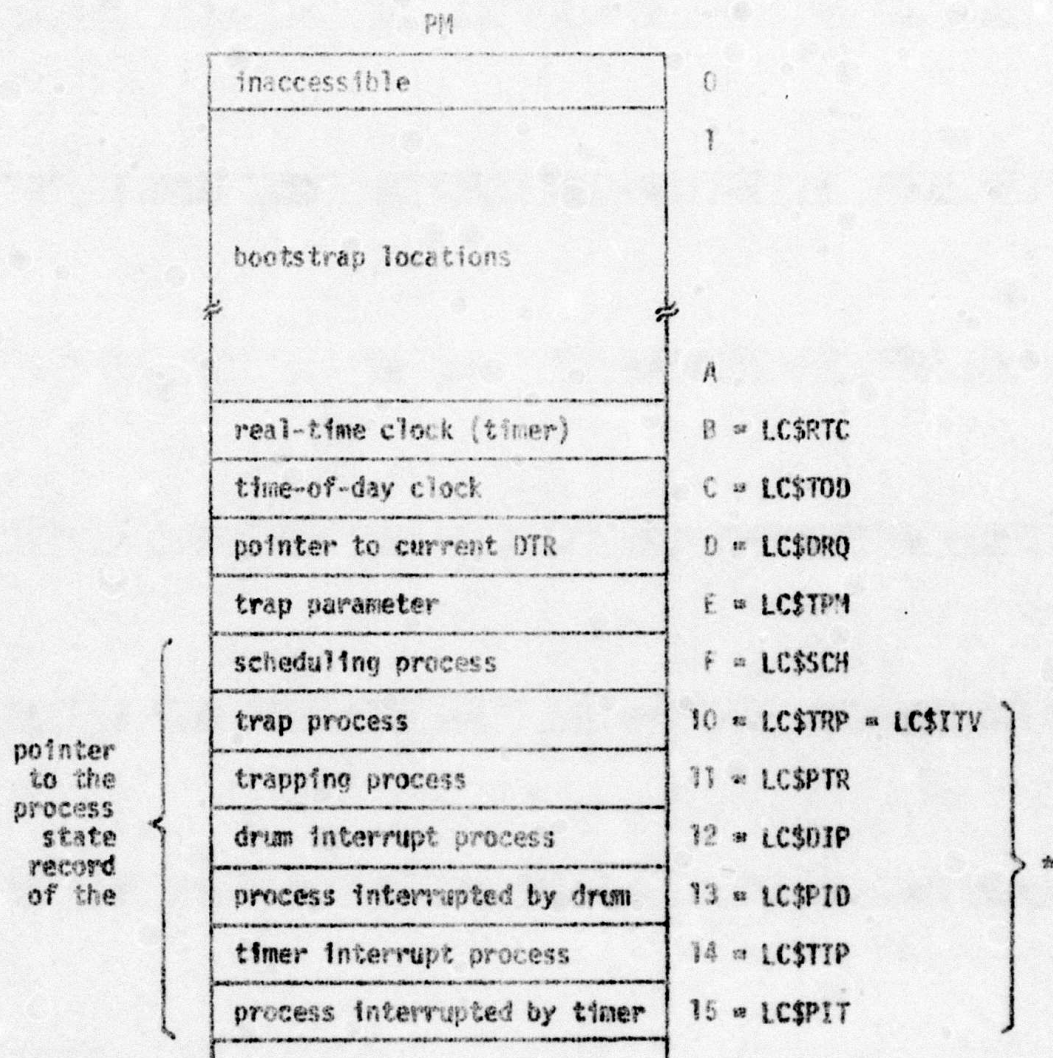
A drum transfer involves the following actions:

- (1) CPU sets up a drum transfer record and updates location D to point to it.
- (2) CPU instructs drum controller to transfer by executing a START DRUM instruction. The execution of this instruction causes a control signal to be sent to the drum controller.
- (3) After receiving the control signal from the CPU, the drum controller checks the drum transfer record pointed to by location D and proceeds to transfer one word at a time.
- (4) After transferring 128 words, the drum controller informs the CPU about the completion of the transfer via an interrupt.
- (5) After the occurrence of the interrupt, the CPU saves the state of the currently running process in the process state record pointed to by the state pointer register SP (i.e. it "interrupts" it) and passes control to the drum interrupt process (drum interrupt routine). This is done by loading the state of the drum interrupt process from the process state record pointed to by a fixed location in primary memory (see next section). The drum interrupt process will record the completed transfer and possibly initiate another one. Thereafter control is returned to the interrupted process by reloading its saved state.

6 LOW CORE

To facilitate communications between hardware and software, it is necessary to adopt certain conventions. For example, the pointer to the current drum transfer record is kept - by convention - in location D. The logic in the drum controller (hardware) knows about this convention, and any program which initiates a transfer must know about this convention too (i.e. the address D must be embedded in the software). Memory locations which serve this hardware/software communications purpose have traditionally been assigned low core addresses to avoid fragmentation.

On COS-2, low core locations are used in hardware/software communications for the following tasks: bootstrapping, keeping time, drum transfer parameter passing, and interrupt handling. Figure 3 depicts the names and contents of those locations. All addresses of these locations as well as their contents (if they are pointers) are physical addresses.



* Interrupt Transfer Vector: this vector consists of two PSR pointers for each of the priority interrupt levels 1, 2, and 3 (traps, drum, and timer). The second PSR pointer is used to note the state of the interrupted (or trapping) process for resumption at a later time. The first PSR pointer describes which process is to be given control at this interrupt level.

All pointers in this figure are physical addresses.

Figure 3: Names and contents of low core locations.

The bootstrap mechanism allows special standalone programs to be loaded from the card reader. This mechanism, which is activated by pressing a button on the console, reads the first eleven cards (JOB card followed by instructions or constants) from the card reader, discards the JOB card, and stores the contents of the other ten cards in locations 1 through A. Thereafter, the program counter PC is set to 1, and the processor enters RUN mode. This mechanism assumes that the first ten words of the standalone program contain code for reading in the remainder of the program. Such code at the beginning of a program to be bootstrapped is called bootstrap loader.

Location B is dedicated as a timer. At the end of the execution of every instruction this location is incremented by one. A timer interrupt occurs when the contents of location B become zero. The timer is programmed by putting into location B the negative number of instruction cycles after which an interrupt signal is desired.

The time-of-day clock is implemented in location C. Like the timer, this location is incremented by one after every instruction. Thus, time-of-day is kept track of in units of instruction cycles. This location must never be written into if the correct time is to be maintained.

As outlined in the previous section, the current drum transfer record is pointed to by location D.

Currently, COS-2 allows three types of traps: memory violation traps, illegal instruction traps, and operating system calls (OS CALL's). When a trap occurs, the hardware writes the following trap parameter into location E:

<u>type</u>	<u>parameter</u>
memory violation	-2
illegal instruction	-1
OS CALL	nonnegative OS CALL number

A pointer to the process state record of the scheduling process is kept in location F.

The interrupt transfer vector starts at location 10. It consists of two words per interrupt level. When more than the current three levels are to be used, additional pairs of words must be reserved.

7 QUEUES

Several COS-2 instructions aid in accessing queues of an arbitrary number of records of arbitrary length. Figure 4 shows the one-word virtual linkage pointers. A negative pointer indicates the last record or an empty queue. It also shows the second words of records; some instructions use them too.

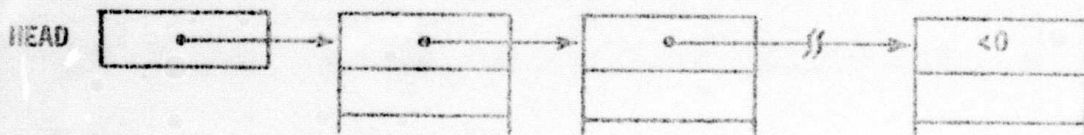


Figure 4: The standard queue format.

OPCODE HEX DEC		MNEMONIC	ADDR. USED	OPERATION
Type 2: Input-output				
20	32	HALT*		HLT=1 (terminates execution)
21	33	READ CARD*		Reads next card into AC and indicates type of card by skips: 0 - JOB card 1 - DATA card 2 - END card 3 - no more cards in hopper 4 - instruction or constant
22	34	PRINT LINE*		Advances one line and prints contents of AC
23	35	HEADER*	A2	prints on line printer the header of the job whose PSR is pointed to by PM(L2)
24	36	START DRUM*		Starts drum controller
25	37	unused		
26	38	unused		
27	39	unused		
28	40	IT RESET*	A2	$IT = IT \wedge PM(L2)$
29	41	AR SET*	A2	$AR = AR \vee PM(L2)$
2A	42	AR RESET*	A2	$AR = AR \wedge PM(L2)$
2B	43	unused		
2C	44	unused		
2D	45	NO OPERATION		Does nothing
2E	46	OS CALL**	A2	$IT = IT \vee TRBIT$; sets trap bit $PM(LC\$TPM) = A2$; OS CALL number
2F	47	unused		

OPCODE		MNEMONIC	ADDR. USED	OPERATION
HEX	DEC			
Type 3: Operating systems support				
30	48	GET RECORD	A2	Does nothing if queue A2 is empty. Otherwise, it removes the first record from queue A2, returns a pointer to it in XR and skips.
31	49	RETURN RECORD	A2,A1	Does nothing if XR does not point to a record in queue A1. Otherwise, it removes from queue A1 the record pointed to by XR, links it as the first record on queue A2 and skips.
32	50	APPEND RECORD	A2	Appends record pointed to by XR to queue A2.
33	51	INSERT RECORD	A2,A1	Inserts record pointed to by XR in queue A2 before the first record whose second word is larger than the second word of the inserted record under the mask PM(L1). If no such record exists, the record pointed to by XR is appended to queue A2.
34	52	FIND RECORD	A2,A1	Returns in XR a pointer to the first record in queue A2 whose second word is equal to AC under the mask PM(L1) and skips if such a record exists. Does nothing otherwise.
35	53	unused		
36	54	SP=XR*		SP=XR
37	55	XR=SP*		XR=SP
38	56	SWITCH PROCESS*	A2	Sets PC to V2 and saves AC, XR, PC, MD and the map in the PSR pointed to by SP. Then XR is copied into SP, and the same registers are loaded from the PSR which SP now points to.
39	57	ALLOCATE	A2	Does nothing if PM(L2) is zero. Otherwise, PM(L2) is scanned from the left until the first 1-bit is found. This bit is cleared, its position is returned in XR and the instruction skips.

OPCODE HEX DEC		MNEMONIC	ADDR. USED	OPERATION
3A	58	DEALLOCATE	A2	Sets in PM(L2) the bit whose position is given in XR and skips if this bit was not already set. Does nothing if this bit was set or if $XR < 0$ or if $XR > 31$.
3B	59	PUNT*	A2	Like HALT, but also indicates system error code A2.
3C	60	GET ABSOLUTE*	A2	$AC = PM(V2)$
3D	61	PUT ABSOLUTE*	A2	$PM(V2) = AC$
3E	62	unused		
3F	63	unused		

RUTGERS UNIVERSITY - Department of Computer Science - CS 416/18
M. Ruschitzka
Spring 1977

CAL -- CAROL
REFERENCE MANUAL

1 INTRODUCTION

The designers of a new computer system usually avail themselves of an operational computer system (the host system, or simply host) for simulation purposes. Both the hardware and the operating system are usually simulated on this host. The hardware simulator may be written in any suitable language available on the host; the proposed operating system will typically be written in a programming language which is expected to be supported on the new system. Assuming that the machine languages of the new system and the host differ, none of the language translators on the host can be used to generate code for the new operating system. Thus, it will be necessary to write a new translator which runs on the host and translates the programming language in which the new operating system is to be written into the new system's machine language. Since this machine language cannot be executed on the host, such a translator is called a crosstranslator. However, the execution of code generated by the crosstranslator can easily be simulated by the hardware simulator running on the host.

When the new operating system has been coded up, it will be crossassembled and loaded into the hardware simulator for debugging. After the major design flaws have been ironed out, the prototype hardware will be implemented. Thereafter, it remains to load the new operating system, which exists as a file in the host, into the prototype system. This can be accomplished by crossassembling it and storing the machine language code on some transferable medium (punched cards, magnetic tape, disk pack, etc.). This medium can now be moved from an I/O device of the host to an I/O device of the prototype system. From there, the operating system can be bootstrapped into the new system.

The IBM 370/168-158 complex serves as the host for the COS development project. In addition to the hardware simulator, a two-pass crossassembler for COS Assembly Language (CAL) has been developed on the host. Its output, COS machine language, can be loaded into the simulator (i.e. the primary memory of the simulator) from the driver which represents the console. The other mode of operation, namely punching object code on cards on the host, transferring them to the hopper of the COS card reader, and bootstrapping from there, can also be simulated. A subroutine package, which combines the Cross Assembler and the card Reader for Object Language (CAROL), has been made available on the host for this purpose. Calling CAROL will yield the next object language card from the card reader. Figure 1 illustrates.

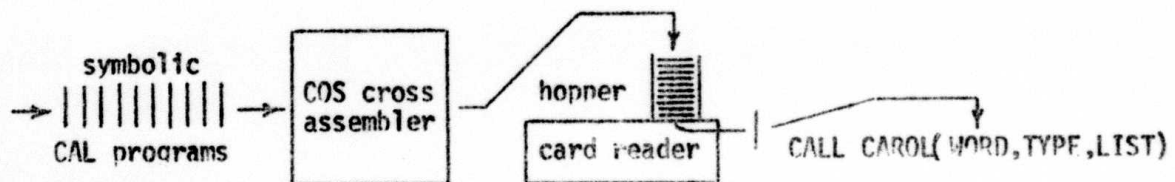


Figure 1: Components and operation of the subroutine package CAROL.

Internally, CAROL maintains a data structure for object code. This structure corresponds to the hopper. Whenever this data structure becomes empty, the COS cross assembler is called and will fill it with the object code and the input data of the next syntactically correct job (syntactically erroneous jobs are flushed). When called, CAROL(WORD,TYPE,LIST) returns the value and the type of the next card in the locations specified by WORD and TYPE respectively. The third parameter, LIST, is an input parameter which is passed on to the COS crossassembler; it specifies listing options. The following options are available:

<u>LIST</u>	<u>listings</u>
0	none
1	program and data
2	program and data, symbol table
3	program and data, symbol table, object code, and data values

In order to distinguish assembler listings from other printout generated by the hardware simulator, the crossassembler precedes and trails all listings with lines of plus signs.

2 COS ASSEMBLY LANGUAGE

A job written in CAL is a collection of statements which represent COS machine instructions, control instructions, pseudo instructions, and data. One statement is written per line or, equivalently, punched per card. The language is blank-insensitive and may be coded in free format. Apart from the pseudo instructions, every statement causes the generation of one machine word.

Typical statements consist of an optional label, a mnemonic, and optional operands followed by a semicolon. Labels are terminated by a colon, and operands are separated by a comma and enclosed in parentheses. The first operand may be subjected to indexing and indirection: this is indicated by the postfixes .X or .I or .I.X for both. Comments may be written after the semicolon or on lines beginning with an asterisk. Valid names are sequences of up to 8 characters which do not begin with a single quote. A single quote indicates a hexadecimal constant (all constants are hexadecimal). Input data are constants followed by a semicolon and possibly a comment.

For most purposes, the informal description of the preceding paragraph and the example in chapter 4 will suffice to characterize CAL. However, sometimes a more formal definition is needed. A variety of metalanguages have been developed to aid in formally defining the syntax of a programming language. Among them, the Backus Normal Form (BNF) has obtained widespread use. The next chapter presents a description of the CAL syntax in BNF.

3 BNF FOR CAL SYNTAX

The following symbols represent the elements of the metalanguage BNF:

- // These metasympols are used as delimiters to enclose the name of a class.
- ::= This metasympol may be read as "is defined as" or "consists of".
- ! This metasympol is read as "or", it separates alternatives.
- [] These metasympols surround optional classes. If the empty class is termed NIL, [/name/] is equivalent to /name/ ! NIL.

Names of classes are self-explanatory. Thus, comments are kept to a minimum.

/job sequence/ ::= /job/ ! /job sequence/ /job/

/job/ ::= /job stmt/ /program/ [/data stmt/ /data/] /end stmt/

/program/ ::= /stmt/ ! /program/ /stmt/

/data/ ::= /datum/ ! /data/ /datum/

/job stmt/ ::= JOB (/const/) ; [/comment/]
/const/ defines the job number which must be within '10 and '37.

/data stmt/ ::= DATA ; [/comment/]

/end stmt/ ::= END [(/ond/)] : [/comment/]
In an /end stmt/ /ond/ may specify the starting location (default is 1).

/stmt/ ::= /cos stmt/ ! /pseudo stmt/

/cos stmt/ ::= [/name/ :] /mnem/ [(/ond/ [.I] [.X] [, /ond/])]; [/comment/]

/mnem/ ::= any COS mnemonic or mnemonic defined by /opcode stmt/
All mnemonics may be shortened to at least four characters.

/pseudo stmt/ ::= /comment stmt/ ! /array stmt/ ! /equate stmt/ !
/const stmt/ ! /opcode stmt/

/comment stmt/ ::= * /comment/

/array stmt/ ::= [/name/ :] ARRAY (/ond/) : [/comment/]
If /ond/ is a /name/, it must have been defined in a preceding /stmt/.

/equate stmt/ ::= EQUATE (/name/ , /const/) : [/comment/]
Assigns the value /const/ to /name/.

/const stmt/ ::= [/name/ :] CONSTANT (/ond/) : [/comment/]

/opcode stmt/ ::= OPCODE (/name/ , /const/) : [/comment/]
Defines new /mnem/ if /name/ is different from existing /mnem/'s and if /const/ is an unused operation code.

/name/ ::= /char /[/char/[/char/[/char/[/char/[/char/[/char/[/char/]]]]]]

/char / ::= any character except /delm/

/delm/ ::= * ! : (' ! . ! , !) ! ;

/opd/ ::= /name/ ! /const/
/name/ must be defined exactly once in the program.

/const/ ::= ' /dia/[/dia/[/dia/[/dia/[/dia/[/dia/[/dia/[/dia/]]]]]]

/dia/ ::= 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9 ! A ! B ! C ! D ! E ! F

/datum/ ::= /comment stmt/ ! /const/ ; [/comment/]

/comment/ ::= any sequence of characters

4 AN EXAMPLE

	JOB	('25); while FREE FORMAT IS POSSIBLE, IT IS MORE
* legible	TO START IN COLUMNS 1, 12, 26 FOR LABELS, MNEMONICS AND OPERANDS.	
	EQUATE	(N, '1B); THIS AND THE NEXT STATEMENT ARE EQUIVALENT
AR:	ARRAY	(N); TO ARRAY ('1B); VALUE OF AR IS ONE.
VARIABLE:	CONSTANT	('FFFFFFF); THIS IS MINUS ONE
VAR2:	CONSTANT	('ABC); COMMENTS ABOVE LOOK MESSY BECAUSE THEY
RESULT:	ARRAY	('1); ARE NOT ALIGNED.
START:	AC LOAD	(VARIABLE); LOAD AC
	XR LOAD	(VAR2); AND XR WITH OPERANDS.
	CALL	(ROUTINE); CALL SUBROUTINE (result IS
	XR=AC;	POINTED TO BY AC - COPY).
	BACKMOVE	('0.X, RESULT); GET IT INTO RESULT.
	HALT;	
ROUTINE:	CONSTANT	('0); SAME AS ARRAY ('1);
	AC STORE	(TEMP1); MOVE THE TWO PARAMETERS
	XR STORE	(TEMP2); INTO MEMORY.
	ADD	(TEMP2, TEMP1); ADD THEM.
	AC LOAD	(POINTER); GET POINTER TO SUM.
	GO TO	(ROUTINE.I); RETURN FROM SUBROUTINE.
TEMP1:	CONSTANT	('8); RESERVES AND INITIALIZES WORD.
TEMP2:	CONSTANT	('AA); GARBAGE IN EITHER CASE.
POINTER:	CONSTANT	(TEMP2); GENERATES RETURN POINTER

* THIS PROGRAM SATISFIES THE REQUIREMENTS OF ASSIGNMENT PROBLEM 2.4

DATA;
* negative INTEGERS FROM -1 to -4
'FFFFFFF;
'FFFFFFE;
'FFFFFFD;
'FFFFFFC;

END

(START);

STARTING LOCATION

RUTGERS UNIVERSITY - Department of Computer Science - CS 416/18
M. Puschitzka
Spring 1977

C O S M O S
R E F E R E N C E M A N U A L

1 INTRODUCTION

The COS Multiprogramming Operating System (COSMOS) is the product of an integrated design methodology which emphasizes the functional structure of the system modules as well as their implementation in the most suitable technology. The complexity of the design and the coding effort can be considerably reduced by moving basic operating systems functions which traditionally have been implemented in software into microcode and/or hardware. In particular, access functions for the major system data structures and the transfer of control from one process to another have been included in the instruction repertoire. The overall organization of COSMOS is defined in terms of levels of abstraction.

The mode of operation of the system is similar to Dijkstra's "THE" Multiprogramming System. Unnecessary sophistication has been sacrificed in order to keep the implementation effort manageable. For the same reason, COSMOS processes only binary information. It relies on a crossassembler (CAROL) for symbol manipulation.

COSMOS is designed to run on a COS-2 configuration. This configuration includes a CPU, primary memory, a drum controller, a drum, a card reader, and a line printer. Memory is paged. The CPU features a priority interrupt system, a clock, a timer, and a console. Special features of the console include a breakpoint facility and a bootstrap mechanism which is coupled with the card reader.

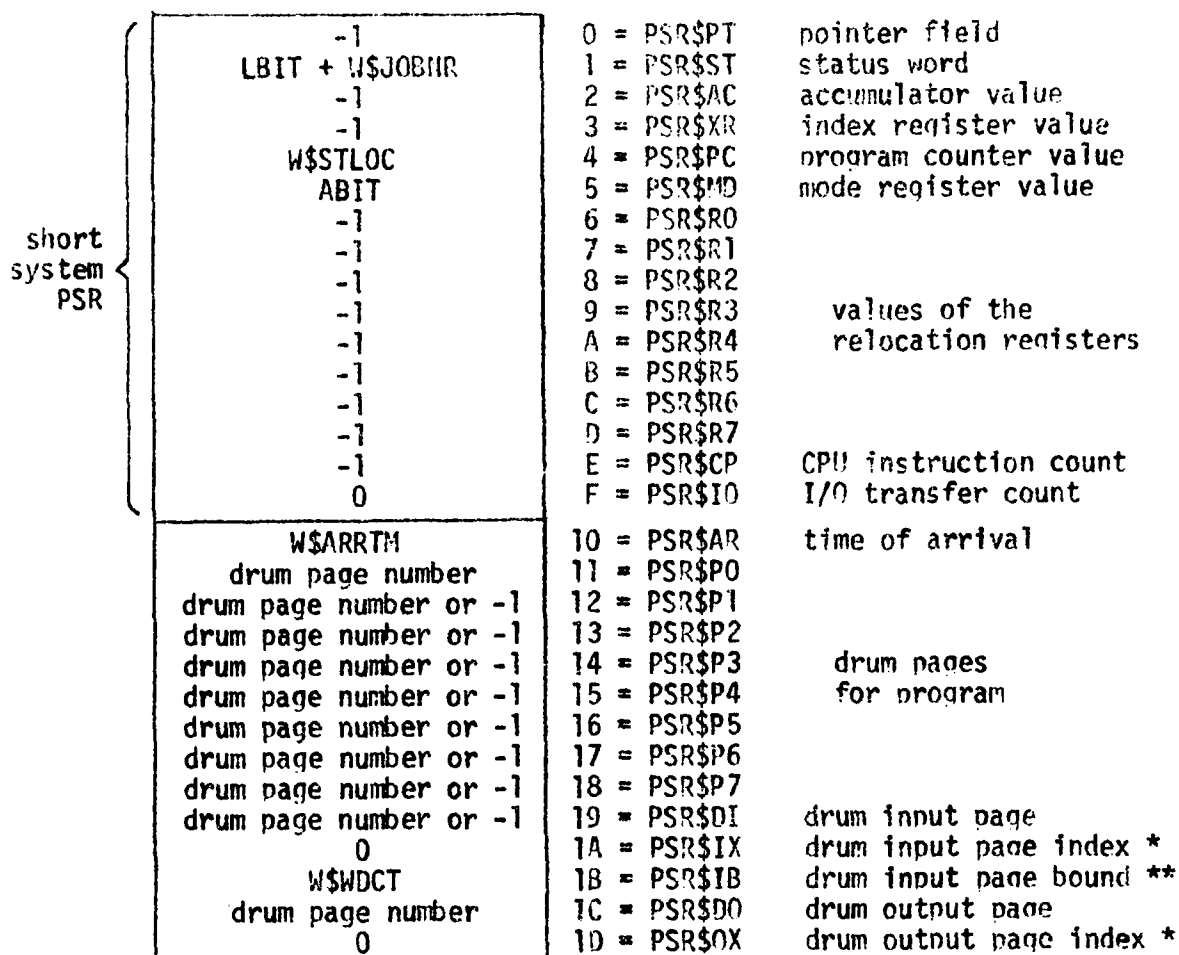
2 MODE OF OPERATION

COSMOS is brought up by bootstrapping from the card reader. After initialization, the system enters the stationary mode in which it accepts user jobs from the card reader for processing. A job must be submitted in the following order: JOB card, program, a DATA card and input data (optional), and another END card. Syntactically incorrect jobs are flushed by the cross-assembler and thus not passed on to the card reader.

At any one time, several jobs may be in the system at various stages of completion. Any one job, however, will be processed in the following sequence. After being spooled to the drum, a job's program is loaded into core and then given control of the CPU. For input, the program may execute an OS CALL which will provide it with the next input word from the input data page on the drum. Another OS CALL causes the transfer of an output word to the output data page on the drum. After termination of a user job, its output is printed on the line printer together with some statistics.

3 COSMOS DATA STRUCTURES

COSMOS maintains the state of a process in a Process State Record (PSR). User PSR's are longer than system PSR's because they also contain spooling information and the state of the input and output data pages. Except for the PSR's of the scheduler and the interrupt processes, all system and user PSR's are chained in increasing order of process numbers (decreasing order of priority) on the queue PSRQ. Figure 1 shows the format of a PSR and its initial values for a spooled-in user process. Figure 2 illustrates in detail the bit allo-



* displacement of next access; ** total number of input data words

Figure 1: Format of a Process State Record and its initial values.

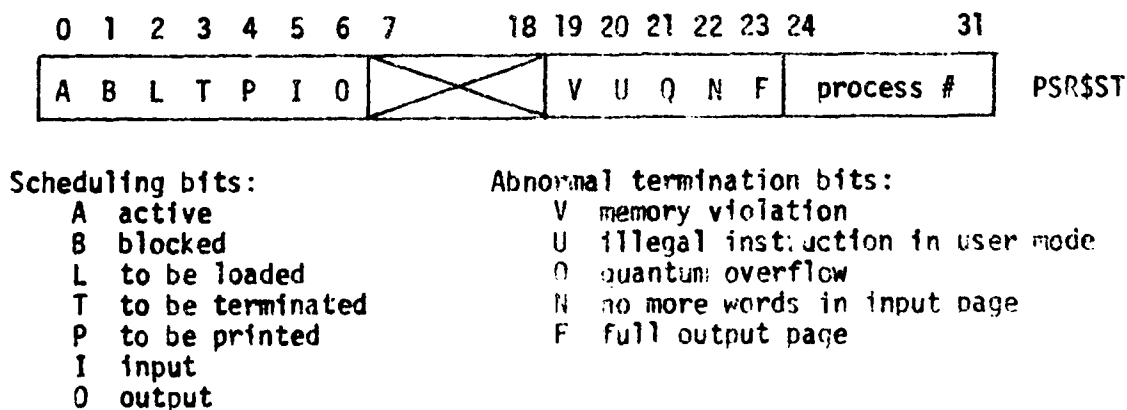
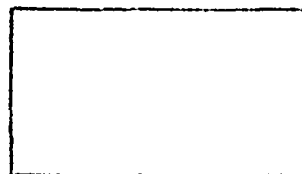


Figure 2: Bit allocation in a PSR status word.



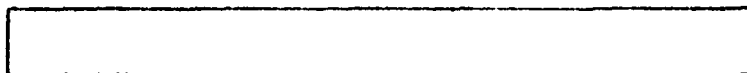
0 = DTR\$PY pointer field
 1 = DTR\$DA drum page number
 2 = DTR\$CA physical core address
 3 = DTR\$RW read/write specification
 4 = DTR\$SP pointer to PSR of requesting process

Read/write specification:

<u>DTR\$RW</u>	<u>specification</u>
≤ 0	read from drum to core
> 0	write from core to drum
-2	read requested by system process
-1	read drum input page of user process
0	read drum output page of user process
1	write drum output page of user process
2	write requested by system process

Figure 3: Format and encoding of a Drum Transfer Record.

0 1 2 3 29 30 31 bit and page number



The status of a particular page is indicated by its corresponding bit:

a zero bit indicates an allocated or non-existing page,
 a one bit indicates an available, unallocated page.

Figure 4: Format and encoding of an allocation word.

cation of the status word in a PSR.

Drum processes are described by Drum Transfer Records (DTR's) which are chained on the drum queue LC\$DRQ and serviced in FIFO fashion. In addition to the three DTR entries required by the drum controller, COSMOS uses the first word as a queue linkage pointer and the last word as a pointer to the PSR of the process requesting the transfer. Figure 3 illustrates.

With the exception of its own code, which is core-resident, COSMOS allocates all core and drum storage dynamically on a page basis. Figure 4 depicts the format and the encoding of the two words which are used to keep track of the allocation state of core and drum pages.

4 COSMOS STRUCTURE

Two global functions of an operating system concern the efficient management of the hardware components and the provision of a convenient interface to the users of a computer system. According to this rough classification, the COSMOS modules may be separated into user support modules and hardware support modules.

The choice of the functional modules for user support is, of course, strongly influenced by the desired mode of operation. For COSMOS, the following functional modules evolved and have been implemented as processes:

<u>Function:</u>	<u>Process name:</u>
Input spooling	RCARD and WCARD
Loading	LOADER
Input/output	FILE SYSTEM
Termination	TERMINATOR
Printing	PRINT
Scheduling	SCHEDULER

Input spooling makes use of double-buffering and consequently two processes evolved for its implementation. Except for the SCHEDULER, all of the processes named above as well as user processes are given control of the CPU by the SCHEDULER; the term schedulable processes refers to this set of processes.

Since the concept of a process is not known to the bare hardware, not all hardware support modules can be implemented as processes. In fact, three different module types have been adopted: processes, subroutines, and instructions. The different technologies of these types are a consequence of the integrated design methodology adopted for COSMOS. It should be pointed out in this context that the term hardware support has a dual meaning. It may refer to the support of hardware components, but it may also mean support of the operating system by a hardware function. The major hardware support modules are listed below.

<u>Function:</u>	<u>Module name and type:</u>
Service of timer interrupts	Timer Interrupt Process
Service of drum interrupts	Drum Interrupt Process
Service of traps	Trap Process
Service of drum transfer requests	Shared Subroutine PUTDRQ
Accessing of PSR's and DTR's	Queue Instructions, SWITCH PROCESS Instr.
Page allocation	Allocation Instructions

The three module types form a technological hierarchy since a process may contain several subroutines and since a subroutine typically consists of several instructions. In addition, the modules themselves form a functional hierarchy due to the nature of their operations. The term levels of abstractions refers to this functional hierarchy.

It was pointed out that operating systems have two interfaces: to the hardware and to the users. In a way, an operating system "maps" the hardware components such that they appear more convenient to the user. For instance, a user program may reference a file by a symbolic name. Internally, the operating system will map this name into a file number which is mapped into a buffer which is mapped into a drum page. Clearly, those modules of an operating system which relate to files are organized in levels such that higher level modules are defined and implemented in terms of lower level modules. This is not only true for the concept of a file, but for all virtual concepts (e.g. virtual memory, operating system calls) provided by an operating system. While the existence of such levels of abstractions is obvious, it is often quite difficult to draw the dividing lines between them. This task is difficult because it requires a thorough understanding of all system aspects. Considering the size of contemporary operating systems, such a complete understanding is often impossible for any single individual. A number of basic criteria aid in the definition of these levels:

- .a module must not be defined in terms of a higher level module
- .a data structure and its access modules should be in the same level
- .a level should implement a concept, i.e. provide an abstraction to the higher levels

Figure 5 displays the levels of abstraction adopted for COSMOS.

Control may be transferred between processes by either of two mechanisms: the SWITCH PROCESS instruction and the interrupt mechanism. The latter serves to forcefully transfer control to an interrupt process. The SWITCH PROCESS instruction is used by the SCHEDULER to transfer control to a ready process. All other user support processes as well as the Trap Process and the Timer Interrupt Process use it to return to the SCHEDULER. Finally, the Drum Interrupt Process uses it to return to the interrupted process. The SCHEDULER has no knowledge of the function of the process it is giving control to; all it checks is whether a process is ready (active and unblocked) to run. It is the responsibility of the other system processes to maintain the scheduling bits of all schedulable processes. Typically, a process will activate another one when it has evidence that the other process has work to do. At the same time an indication must be given as to what quantity (e.g. a user process) the other process should work on. A process is deactivated by another process or by itself when it is evident that no more productive work can be done.

LEVEL	ABSTRACTIONS	FUNCTIONS	MODULES	DATA STR.
7		user problem execution		
6	virtual user memory, files, OS CALL's	user support (spooling, I/O, loading, termination, printing)	RCARD, WCARD, LOADER, FILE SYSTEM, TERMINATOR, PRINT	
5	CPU multi-programming	CPU allocation	SCHEDULER	
4	real-time independence	service of interrupts and traps	Timer Interrupt Process, Drum Interrupts Process, Trap Process	
3	drum multi-programming	allocation of drum controller	PUTDRQ	
2	CPU and drum processes	accesses of PSR's and DTR's	Queue instructions, SWITCH PROCESS, START DRUM	PSR queue, drum queue
1	storage allocation	accesses of allocation words	ALLOCATE, DEALLOCATE	core and drum allocation words

Figure 5: Levels of abstractions in COSMOS.

5 COSMOS SOFTWARE

The COSMOS software consists of the code of the ten system processes and the subroutine PUTDRQ. In order to reflect the system structure in the code, naming conventions for variables and labels have been adopted. All names of local quantities are prefixed with characters indicating the module to which they belong. Thus, names of shared global quantities are characterized by the lack of a prefix. Further particulars of a process module include a process number, a range of PUNT numbers, and a global variable containing its PSR address. The particulars of the various process modules are summarized below:

	<u>prefix</u>	<u>proc. #</u>	<u>PUNT #'s</u>	<u>PSR ptr.</u>
<u>Non-schedulable Processes:</u>				
SCHEDULER	SS	'0	'10-'1F	LC\$SCH
Trap Process	TR\$	'0	'20-'2F	LC\$TRP
Drum Interrupt Process	DI\$	'0	'30-'3F	LC\$DIP
Timer Interrupt Process	TIS	'0	'40-'4F	LC\$TIP
<u>Schedulable Processes:</u>				
RCARD	R\$	'88	'50-'5F	ADPSRR
WCARD	W\$	'89	'60-'6F	ADPSRW
(shared spooling variables)	RW\$			
LOADER	L\$	'87	'70-'7F	ADPSRL
FILE SYSTEM	F\$	'85	'80-'8F	ADPSRF
TERMINATOR	T\$	'5	'90-'9F	ADPSRT
PRINT	P\$	'7	'A0-'AF	ADPSPP

Initially, the PSR's of the schedulable system processes are chained on the PSR queue PSRQ in increasing order of their process numbers, or equivalently in decreasing order of priority (the equivalence is due to the simple scheduling strategy adopted for the SCHEDULER which scans the PSR queue until it finds the first process ready to run).

5.1 SCHEDULER

The scheduler serves two functions: dynamic initialization of the system and scheduling. The initialization sequence is given control from the bootstrap loader and involves completing the bootstrap operation and the proper setting of the CPU registers.

The scheduling sequence attempts to find a ready process in the PSR queue and pass control to it after setting the timer to prevent infinite looping. If no ready process is found, the scheduler makes sure that there are still active processes in the PSR queue and branches back to the beginning of the scheduling sequence. When all processes are deactivated, the batch of user jobs submitted to COSMOS must have been serviced to completion and the scheduler HALTs.

Code for SCHEDULER:

```

*****
*   SCHEDULER   *
*****
*
*   INITIALIZATION (GETS CONTROL FROM BOOTSTRAP LOADER)
*
S$INIT:  XR LOAD      (LC$SCH);
         SP=XR;
         IT RESET    (NEGONE);
         AR RESET    (NEGONE);
         AR SET      (INTBITS);
         AC LOAD     (ABIT);
                                INITIALIZE SP
                                CLEAR IT
                                CLEAR AR
                                ENABLE AND AR1 LEVELS
                                INIT PATTERN FOR MATCHING
*
*   SCHEDULING
*
S$ENTRY:  MOVE        (LC$RTC, S$QUANT);
         FIND RECORD  (PSRQ, ABBITS);
         GO TO        (S$ACHECK);
         SWITCH PROCESS (S$ENTRY);
                                SET TIMER
                                FIND READY PROCESS
                                NO PROCESSES READY
                                PASS CONTROL TO READY PROC
*
*   CHECK FOR BATCH COMPLETION
*
S$ACHECK: FIND RECORD  (PSRQ, ABIT);
         HALT;
         GOTO          (S$ENTRY);
                                FIND AN ACTIVE PROCESS
                                NONE, BATCH COMPLETED
                                OBVIOUSLY BLOCKED, KEEP
                                SCANNING PSRQ WHILE WAITING FOR ANY
                                PROCESS TO BECOME READY
*****
*   LOCAL VARIABLES
*
S$QUANT:  CONSTANT    ('FFFFFFF);
                                LARGE NEGATIVE QUANTUM
*****

```

5.2 PUTDRQ

The shared subroutine PUTDRQ serves as the central mechanism for submitting I/O requests to the drum system. When called, PUTDRQ expects in AC a pointer to a 5-word I/O parameter table whose format is identical to a DTR. It copies the I/O parameter table into a DTR which it obtains from the freelist, appends the DTR to the drum queue LC\$DRQ, starts the drum if and only if the drum queue was empty, and returns by skipping. A no-skip failure return indicates that currently there are no DTR's available on the freelist.

Code for PUTDRQ:

```

*****
*   SHARED SUBROUTINE PUTDRQ   *
*****
PUTDRQ:  ARRAY      ('1);          FOR RETURN ADDRESS
        XR STORE    (XRSAVE);
        AC STORE    (ADTABLE);
        GET RECORD  (DTRFRLS);
        GO TO       (PUTDRQ.I);    FAILURE RETURN
        XR STORE    (ADDTR);       DTR ADDRESS
        XR LOAD     (ONE);
XFER:    AC LOAD     (ADTABLE.I.X);
        AC STORE    (ADDTR.I.X);
        LOOP        (DTRSIZE, XFER);

*
*   START DRUM CONDITIONALLY
*
        XR LOAD     (ADDTR);
        AR RESET    (DCBIT);       WATCH OUT FOR DRUM INTER-
*                                     RUPT PROCESS, LC$DRQ IS SHARED!
        APPEND RECORD (LC$DRQ);
        IF <0 GO TO (LC$DRQ.I, RESTORE);
        START DRUM;
RESTORE: AR SET      (DCBIT);
        XR LOAD     (XRSAVE);
        AC LOAD     (ADTABLE);
        ADD         (PUTDRQ, ONE);  IN ORDER TO SKIP
        GO TO       (PUTDRQ.I);    SUCCESS RETURN
*****
*   LOCAL VARIABLES
*
XRSAVE:  ARRAY      ('1);          SAVE LOCATION
ADTABLE: ARRAY      ('1);          ADDR OF I/O PARAMETER TABLE
ADDTR:   ARRAY      ('1);          DTR ADDRESS
*****

```

5.3 INPUT SPOOLING

A user job submitted via the card reader is transferred to the drum before it is loaded for execution. COSMOS employs a dynamic double-buffering scheme for this spooling task. In this scheme, the process RCARD may fill a dynamically allocated buffer A concurrently with the transfer of the previously filled buffer B to the drum. The transfers are initiated and terminated by the process WCARD which also initializes a PSR for the entering job. A set of shared variables prefixed by RW\$ serves for communications between RCARD and WCARD. Via this set, RCARD informs WCARD about the address of the dynamically allocated buffer as well as various job parameters determined during reading (job number, starting location, buffer contents: program or data, number of input words, arrival time).

Consecutive activations of PCARD and WCARD must necessarily concern an alternating sequence of the two buffers: A, B, A, B, A, B, ... The current buffer may be determined by local variables R\$SWITCH and W\$SWITCH which assume the values 0 and -1 (initially 0) and are flipped after every invocation of the corresponding process. Since the two buffers may contain parts of two different user jobs (e.g. the input data of one job in one buffer and the first program page of the next job in the other buffer), the set of shared variables prefixed by RW\$ should be duplicated. If two consecutive words are allocated for each of these variables, indexing with the value of R\$SWITCH or W\$SWITCH may be used to access the currently relevant variable.

The indexed shared variables RW\$TYPE describe the contents of the current buffer. They may assume the values -1, 0, +1 for input data, an empty buffer, and code respectively. RW\$LAST indicates whether the current buffer is the last one of a job (RW\$LAST=1) or not (RW\$LAST=0).

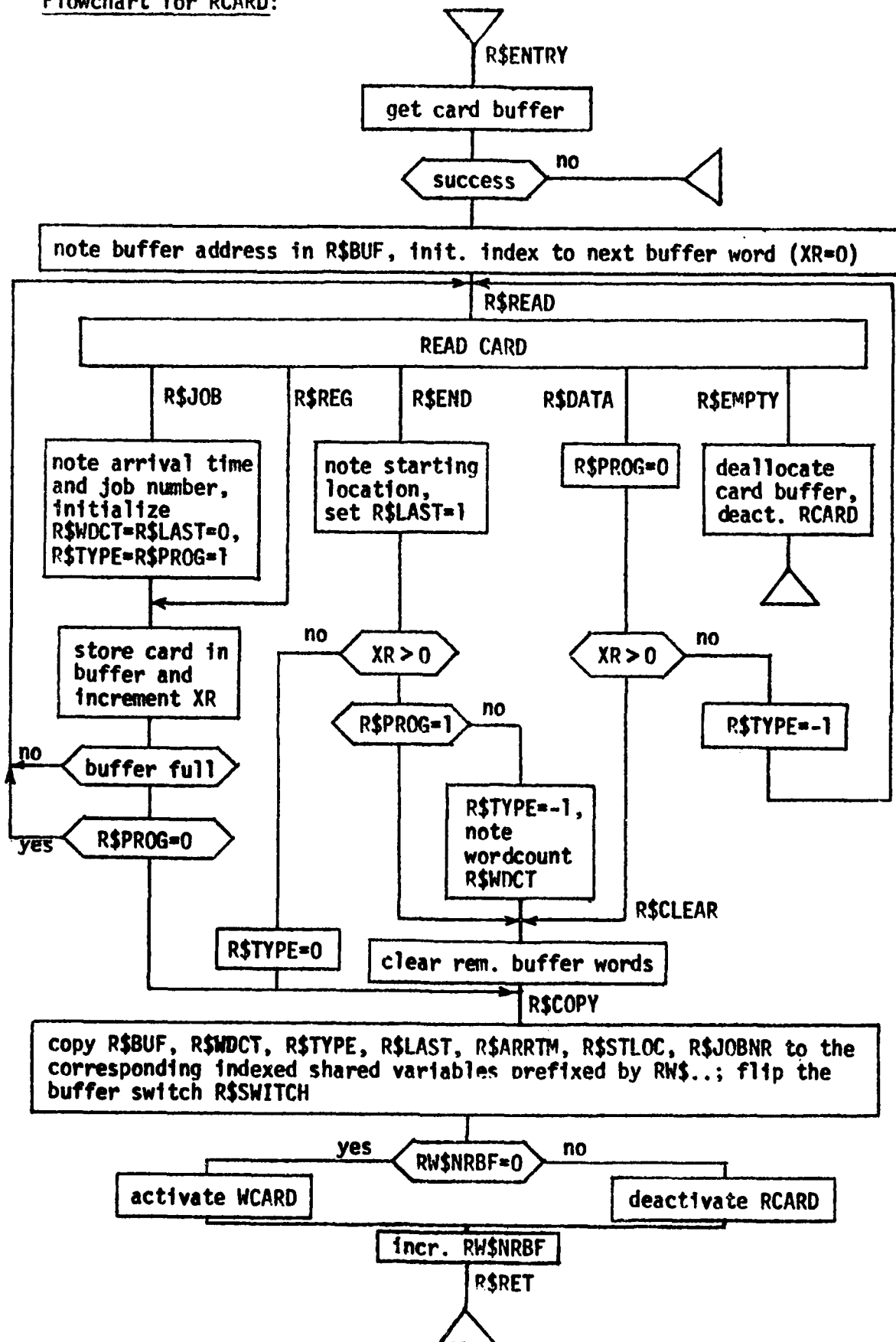
Another shared variable RW\$NRBF indicates the number of buffers (0, 1, 2) currently filled. It is used to control the activations of the two spooling processes. RW\$NRBF will effectively synchronize the activations such that no more than two buffers can be allocated to input spooling at any one time. Since the value of RW\$NRBF is not related to the contents of the buffer(s), RW\$NRBF need not be duplicated.

5.3.1 RCARD

When invoked, RCARD allocates a new buffer. Thereafter, it loops reading cards into this buffer until either an END or DATA card is encountered, the buffer is completely filled, or there are no more cards in the hopper. Upon detection of a JOB card, RCARD notes the arrival time (R\$ARRTM) and the job number (R\$JOBNR) and initializes R\$TYPE, R\$LAST, the wordcount of the input data (R\$WDCT), and R\$PROG. R\$PROG is set to zero when a DATA card is read in (contrary to the JOB card, the DATA card is not deposited in the buffer). Reading an END card will cause the starting location (R\$STLOC) to be noted and, if input data were attached to the job (R\$PROG=0), their number is also remembered (P\$WDCT).

A buffer is shipped to WCARD when it is full or when an END or DATA card is detected (a DATA card arriving as the first card after an invocation and an END card following exactly 128 input data are treated as special cases). Before shipping, the shared variables prefixed with RW\$ are updated from local variables in order to inform WCARD about the nature of the current buffer. Then P\$SWITCH is flipped and RW\$NRBF is incremented. Before returning to the SCHEDULER, RCARD deactivates itself if both buffers are filled or if there are no more cards in the hopper. If both buffers had been empty before the current one was filled, WCARD is activated to work on this current buffer.

Flowchart for RCARD:



Code for RCARD:

```
*****
*   RCARD   *
*****
R$ENTRY:  ALLOCATE      (PMWORD);
          GO TO        (R$RET);
          XR STORE      (R$BUF);
          MULTIPLY      (R$BUF, PGSZ);
          XR CLEAR;

*
*   READ AND PROCESS NEXT CARD
*
R$READ:   READ CARD;
          GO TO        (R$JOB);
          GO TO        (R$DATA);
          GO TO        (R$END);
          GO TO        (R$EMPTY);
          GO TO        (R$REG);
R$JOB:    MOVE          (R$ARRTM, LC$TOD);
          AC STORE      (R$JOBNR);
          MOVE          (R$WDCT, ZERO);
          MOVE          (R$LAST, ZERO);
          MOVE          (R$TYPE, ONE);
          MOVE          (R$PROG, ONE);
R$REG:    PUT ABSOLUTE  (R$BUF.I.X);
          LOOP          (PGSZ, R$READ);
          IF =0         (R$PROG, R$READ);
          GO TO        (R$COPY);
R$DATA:   MOVE          (R$PROG, ZERO);
          XR STORE      (R$XR);
          IF >0        (R$XR, R$CLEAR);
          MOVE          (R$TYPE, NEGONE);
          GO TO        (R$READ);
R$EMPTY:  DIVIDE        (R$BUF, PGSZ);
          XR LOAD       (R$BUF);
          DEALLOCATE    (PMWORD);
          PUNT          ('50);
          XR LOAD       (ADPSRR);
          AND           (PSR$ST.X, CABIT);
          GO TO        (R$RET);
R$END:    AC STORE      (R$STLOC);
          MOVE          (R$LAST, ONE);
          XR STORE      (R$XR);
          IF >0        (R$XR, R$WHAT);
          MOVE          (R$TYPE, ZERO);
          GO TO        (R$COPY);
R$WHAT:   IF >0        (R$PROG, R$CLEAR);
          MOVE          (R$TYPE, NEGONE);
          XR STORE      (R$WDCT);
R$CLEAR:  SUBTRACT      (R$XR, PGSZ);
          IF =0        (R$XR, R$COPY);
          AC CLEAR;
R$CLLP:   PUT ABSOLUTE  (R$BUF.I.X);
          LOOP          (PGSZ, R$CLLP);

*
*   UPDATE VARIABLES SHARED WITH WCARD
*
R$COPY:
```

Code for RCARD (continued):

```
R$COPY:  XR LOAD      (R$SWITCH);
          MOVE        (RW$BUF.X, R$BUF);
          MOVE        (RW$WDCT.X, R$WDCT);
          MOVE        (RW$TYPE.X, R$TYPE);
          MOVE        (RW$LAST.X, R$LAST);
          MOVE        (RW$ARRTM.X, R$ARRTM);
          MOVE        (RW$STLOC.X, R$STLOC);
          MOVE        (RW$JOBNR.X, R$JOBNR);
          COMPLEMENT  (R$SWITCH, R$SWITCH);
```

*

* SYNCHRONIZATION TO ENFORCE AT MOST TWO BUFFERS

*

```
          IF =0      (RW$NRBF, R$ACTW);
R$DEACT:  XR LOAD    (ADPSRR);
          AND        (PSR$ST.X, CABIT);
          GO TO      (R$INC);
R$ACTW :  XR LOAD    (ADPSRW);
          OR         (PSR$ST.X, ABIT);
R$INC:    ADD        (RW$NRBF, ONE);
R$RET:    XR LOAD    (LC$SCH);
          SWITCH PROC (R$ENTRY);
```

* LOCAL VARIABLES

*

```
R$SWITCH:  CONSTANT  ('0);
R$BUF:     ARRAY     ('1);
R$WDCT:    ARRAY     ('1);
R$TYPE:    ARRAY     ('1);
R$LAST:    ARRAY     ('1);
R$ARRTM:   ARRAY     ('1);
R$STLOC:   ARRAY     ('1);
R$JOBNR:   ARRAY     ('1);
R$PROG:    ARRAY     ('1);
R$XR:      ARRAY     ('1);
```

* VARIABLES SHARED WITH WCARD

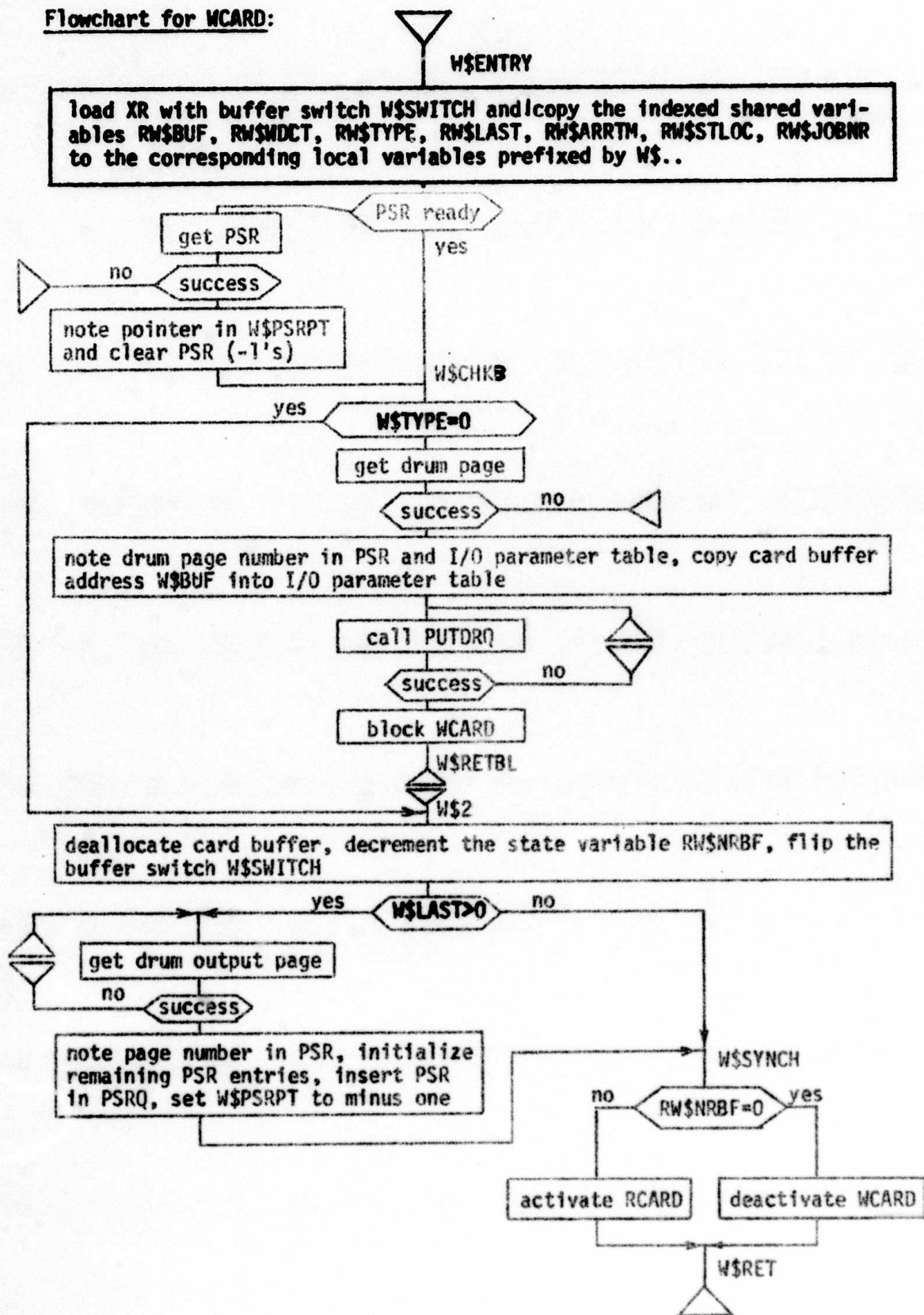
*

```
RW$NRBF:   CONSTANT  ('0);
          ARRAY      ('1);      THIS IS RW$BUF[-1]
RW$BUF:    ARRAY      ('2);
RW$WDCT:   ARRAY      ('2);
RW$TYPE:   ARRAY      ('2);
RW$LAST:   ARRAY      ('2);
RW$ARRTM:  ARRAY      ('2);
RW$JOBNR:  ARRAY      ('2);
RW$STLOC:  ARRAY      ('1);
```

5.3.2 WCARD

When invoked to initiate the transfer of the first buffer of an entering user job, WCARD obtains and initializes a PSR for this user. This PSR is updated during consecutive invocations until the complete job resides on the drum and is ready to be loaded. WCARD supervises one buffer transfer per invocation. If the current buffer is not empty, a drum page is allocated and its address

Flowchart for WCARD:



is noted in the appropriate PSR slot for the given program or input data page. Before calling PUTDRQ, this address and the buffer address are inserted in the I/O parameter table. Thereafter, WCARD blocks itself and will not resume execution until the Drum Interrupt Process unblocks it after the completion of the transfer. Then, the buffer is deallocated, RW\$NRBF is decremented, and RW\$switch is flipped to keep track of the alternating buffer sequence.

If the current buffer is the last one of a job, the drum output page is allocated, the remaining PSR entries are initialized (Figure 1), and the LOADER is activated. Before returning to the SCHEDULER, WCARD activates RCARD if both buffers had been filled before the current transfer. If both buffers are empty after the current transfer, WCARD will deactivate itself.

5.4 DRUM INTERRUPT PROCESS

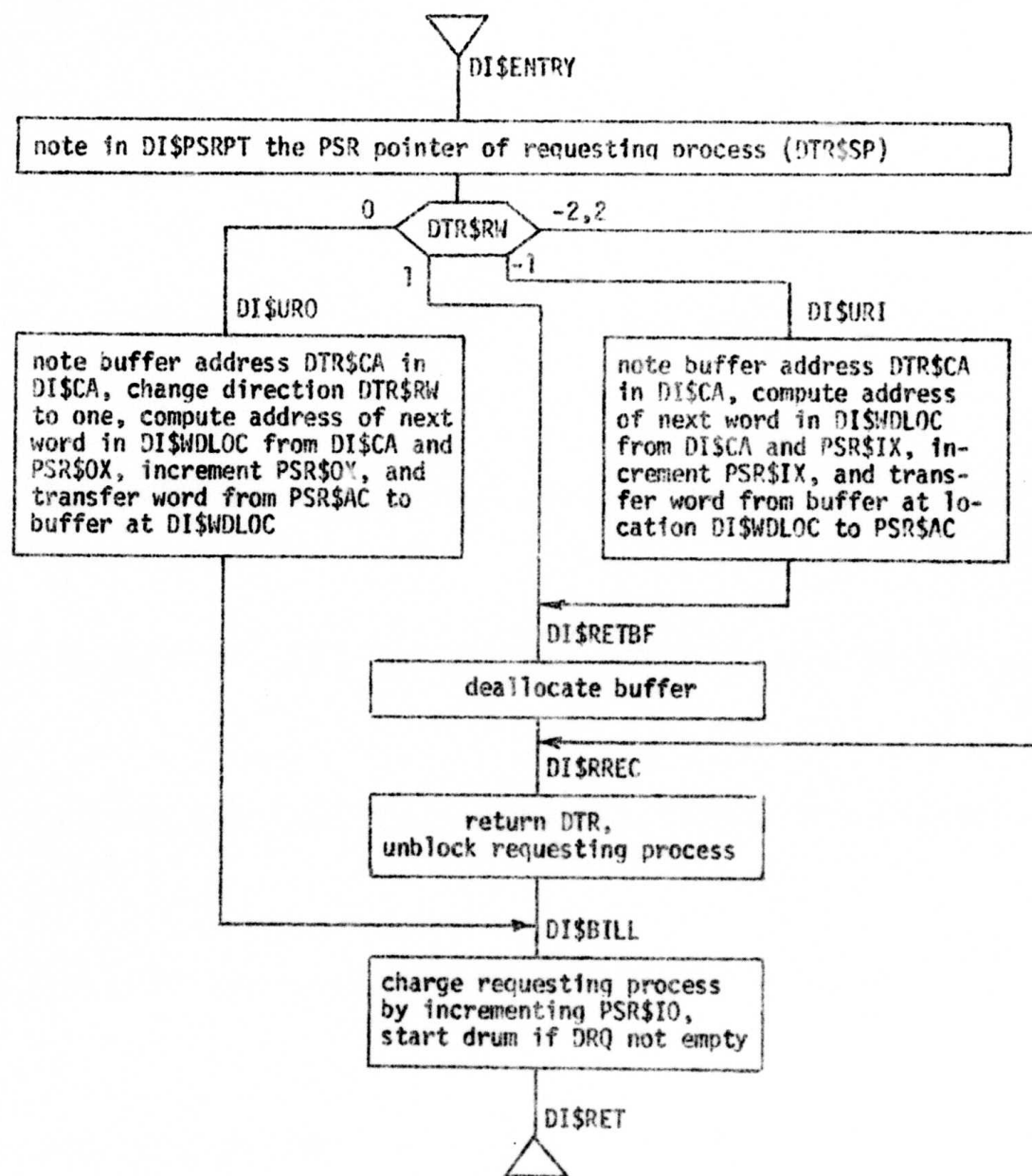
After the occurrence of a drum completion interrupt, the Drum Interrupt Process determines from the current DTR whether the transfer was originally requested by a system process or a user process. In the former case, the Drum Interrupt Process cleans up after the transfer by returning the current DTR to the freelist, unblocking and charging the requesting process, and - if the drum queue is not empty - starting the drum.

If the requesting process was a user process, the Drum Interrupt Process follows up on the actions initiated by the FILE SYSTEM. Logically, these actions do not belong to the Drum Interrupt Process, but they are short and a further activation of the FILE SYSTEM appears cumbersome. Note this as an example of bad system design! Logical cleanliness should be given priority over efficiency considerations unless some very convincing arguments to the contrary have been thought through.

On user input, the next input word is copied from the I/O buffer to the user's PSR, the sequential I/O count is incremented, and the I/O buffer is deallocated. Thereafter, the Drum Interrupt Process cleans up as for a transfer requested by a system process.

For user output, the drum output page must be read into an I/O buffer, updated, and written back to the drum. After it has been read, the Drum Interrupt Process transfers the output word from the PSR to the I/O buffer, updates the sequential I/O count, changes the direction of the current DTR to reflect that the output page is to be written back, and starts the drum after charging the user process for the I/O transfer. The next interrupt, which signals that the drum output page has been written back, is serviced by deallocating the I/O buffer and cleaning up as for a transfer requested by a system process.

Flowchart for Drum Interrupt Process:



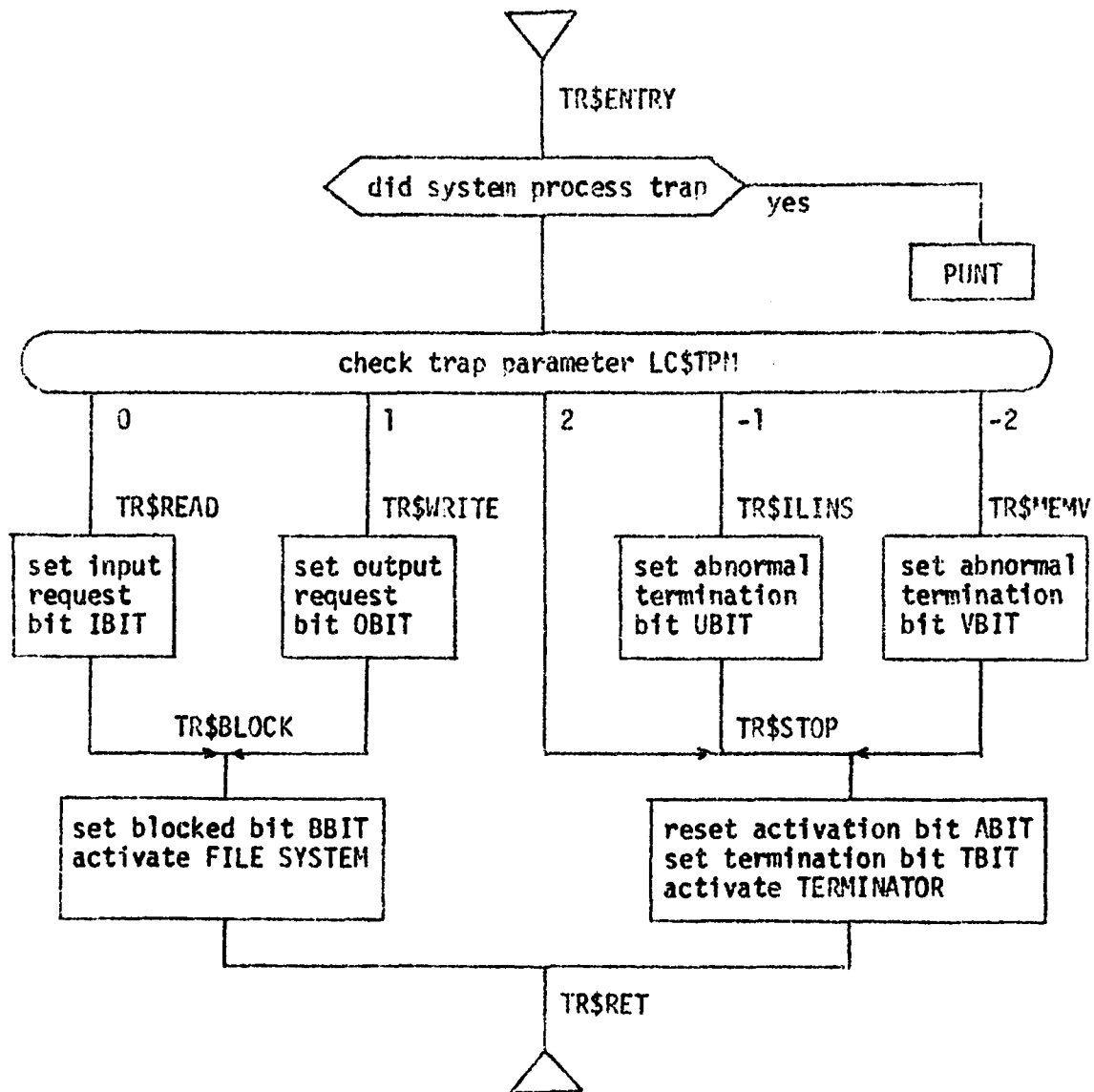
5.5 TRAP PROCESS

The execution of an illegal instruction, a memory violation, or an OS CALL (READ, WRITE, STOP) results in an invocation of the Trap Process. COSMOS itself should not cause any traps, and therefore the Trap Process PUNTS if the trapping process is a system process. For user processes traps represent the means by which control may be passed back to the system which will perform some task for this process. The Trap Process essentially activates a system process to perform this task, specifies the nature of the task in the status word of the user's PSR, and returns to the SCHEDULER.

For READ (WRITE), the scheduling bit IBIT (OBIT) is set in the status word, the user process is blocked, and the FILE SYSTEM is activated to initiate the I/O operation.

After an illegal instruction trap or a memory violation trap, an abnormal termination bit is set in the status word of the user's PSR. Then, the user process is deactivated, scheduled for termination, and the TERMINATOR is activated. The latter actions are also performed when the user process simply STOPS (normal termination).

Flowchart for Trap Process:



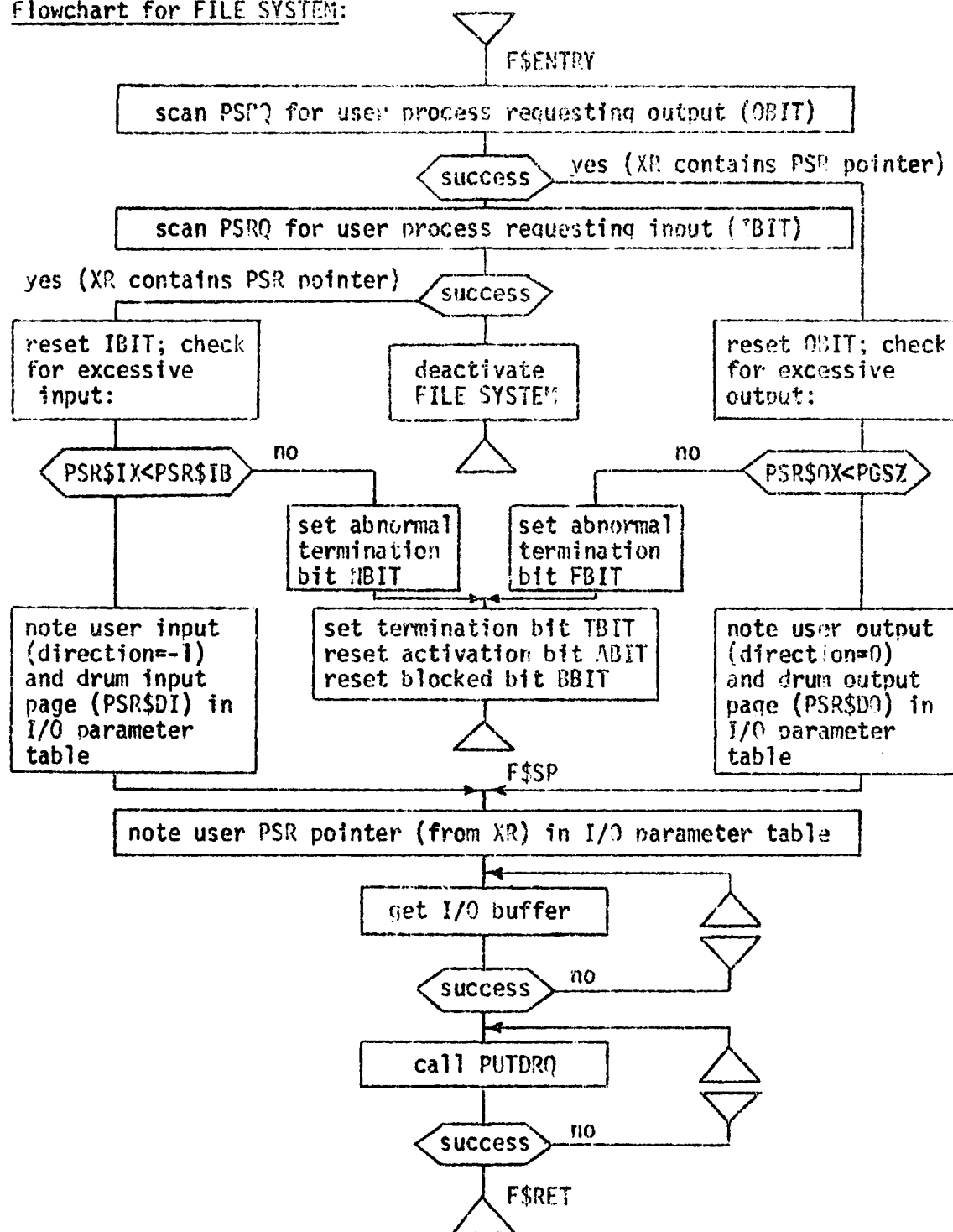
5.6 FILE SYSTEM

The FILE SYSTEM is activated by the Trap Process when a user process executes a READ or a WRITE. Since the Trap Process also sets the scheduling bits IBIT or OBIT, the FILE SYSTEM can find the user process by scanning the PSR queue. The FILE SYSTEM deactivates itself if none of these scheduling bits are set in any of the schedulable processes.

After the requesting user process has been found, a check is performed to make sure that the input data have not been exhausted (for READ) or that the output page is not full (WRITE). If such a condition exists, the appropriate abnormal termination bit is set and the user process is deactivated, unblocked, and scheduled for termination.

If the I/O request is reasonable, an I/O parameter table is set up, an I/O buffer is allocated, and the transfer of the input or output page is initiated by calling PUTDRQ. Since the remaining actions will be performed by the Drum Interrupt Process, the FILE SYSTEM has completed its task and returns to the SCHEDULER.

Flowchart for FILE SYSTEM:



5.7 LOADER

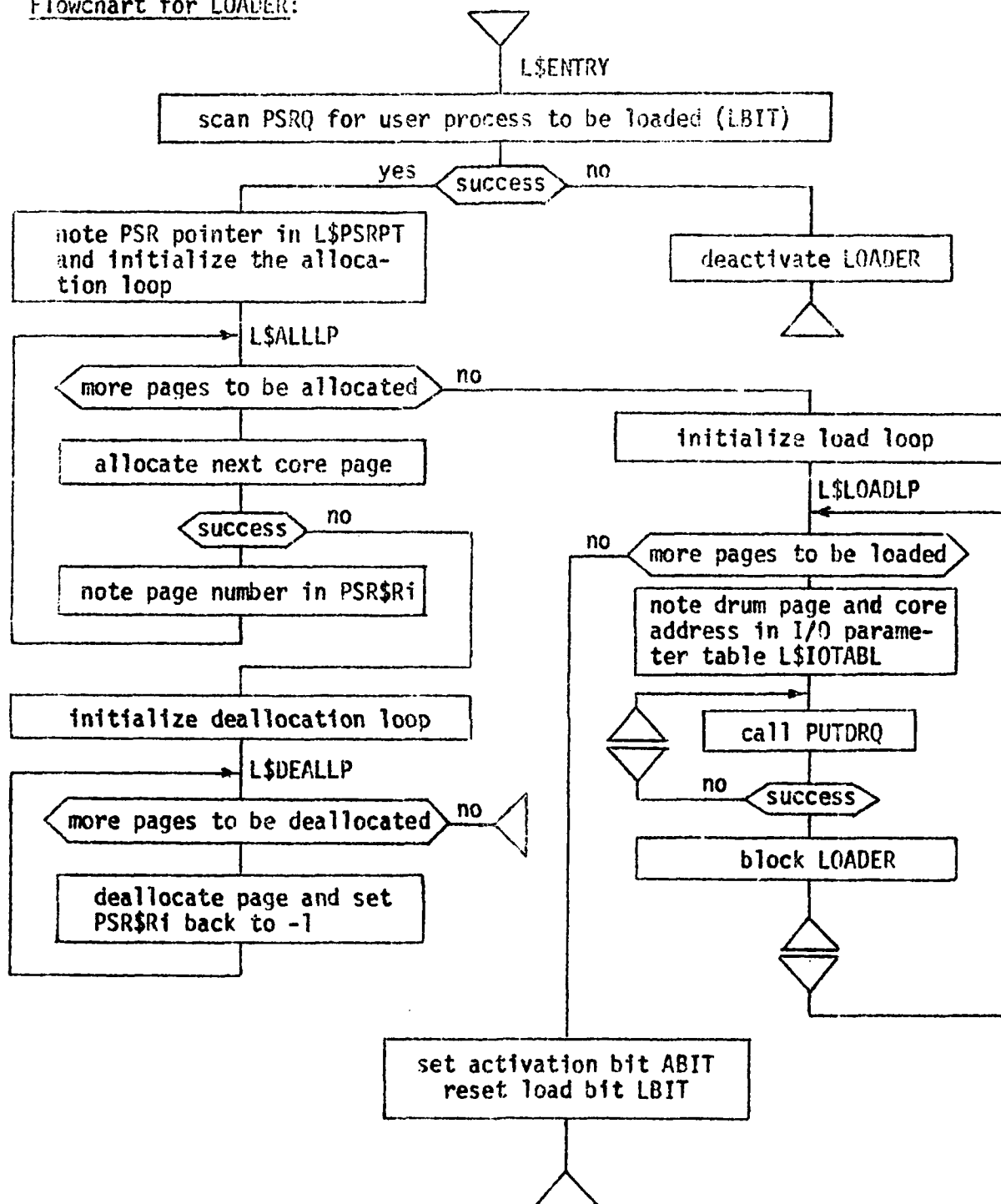
After a user job has been transferred to the drum by the input-spooling system, WCARD schedules the user process for loading and activates the LOADER. When invoked, the LOADER finds the process to be loaded by scanning the PSR queue. The LOADER deactivates itself when there are no more processes to be loaded.

COSMOS employs a static preloading scheme, i.e. all pages of a user process must be loaded before execution may begin. A partially loaded process will therefore tie up core pages without being able to progress towards completion. In order to avoid such an unproductive drain on an important system resource, the LOADER ascertains that there are enough core pages available to load the entire program before it proceeds to load the first program page.

The LOADER consists of three major loops. In the allocation loop, an attempt is made to allocate all necessary core pages. If this attempt fails, those pages which were available and were allocated temporarily are returned to the freelist in the deallocation loop. The LOADER will then return to the SCHEDULER with the intent to try again at a later time.

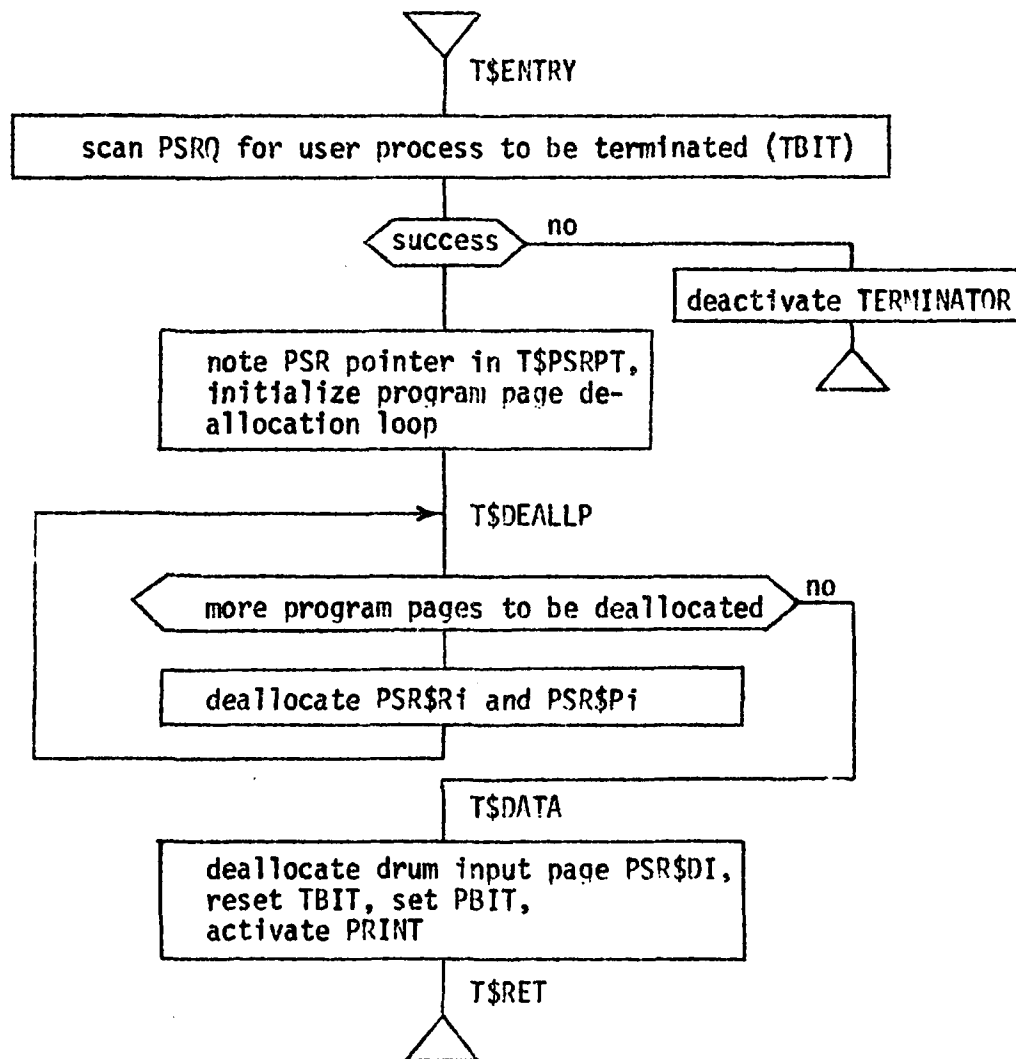
If all necessary pages can be allocated, the load loop is entered. Here, the LOADER sets up an appropriate I/O parameter table, initiates the transfer by calling PUTDRQ, and blocks itself before returning to the SCHEDULER. After the completion of the transfer, the Drum Interrupt Process unblocks the LOADER which continues the load loop until all program pages are in core. Thereafter, the user process is activated (it is now ready to run) and its LBIT is reset. Having successfully completed its task, the LOADER returns to the SCHEDULER.

Flowchart for LOADER:



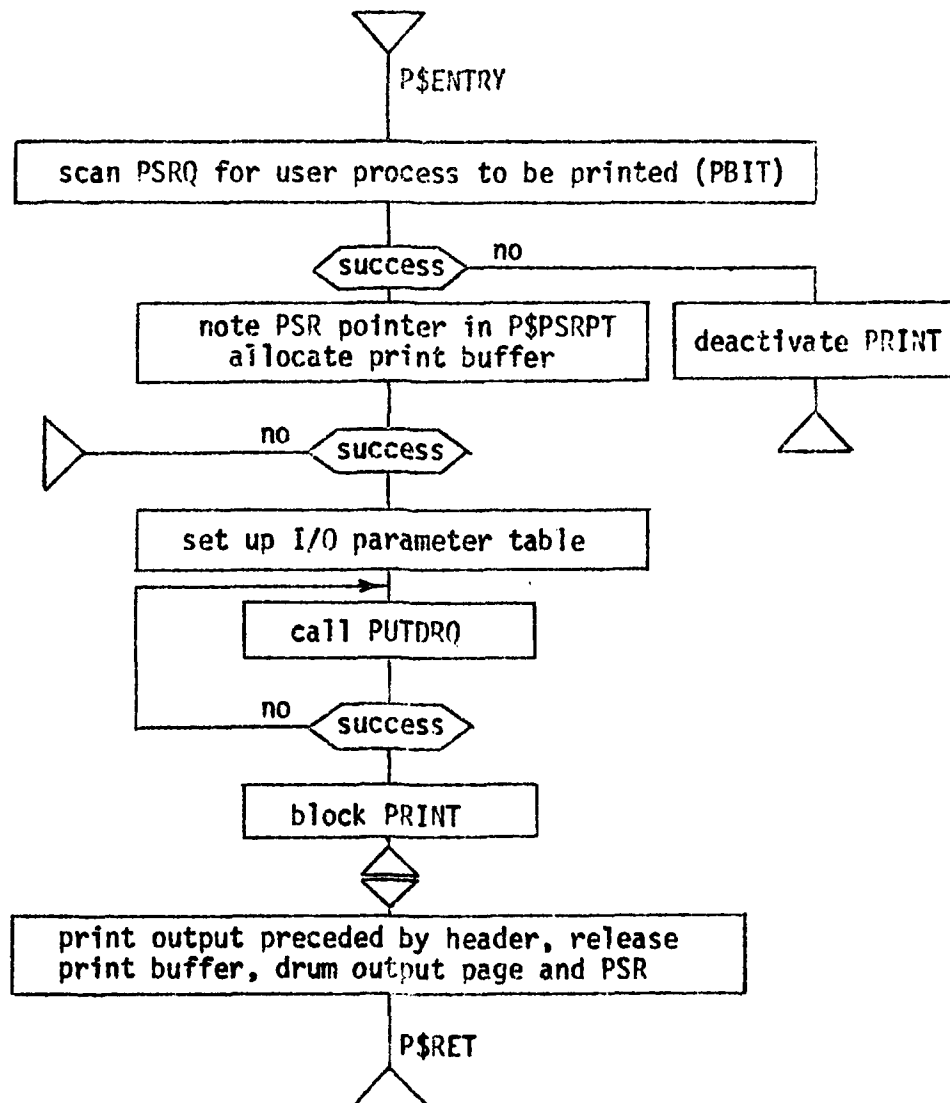
5.8 TERMINATOR

The TERMINATOR is activated by the Trap Process which schedules a user process for termination after a STOP, an illegal instruction trap or a memory violation trap. The TERMINATOR finds the job to be terminated by scanning the PSR queue. It deactivates itself if there are no processes to be terminated. When a process to be terminated is found, all its core pages and all its drum pages except for the drum output page are deallocated. After its TBIT is reset, the user process is scheduled for printing and PRINT is activated. The TERMINATOR returns to the SCHEDULER.



5.9 PRINT

PRINT is activated by the TERMINATOR when it schedules a terminated user process for printing. PRINT finds the user process by scanning the PSR queue or deactivates itself in case there are no more user processes to be printed. After finding a user process to be printed, PRINT allocates a print buffer, sets up an I/O parameter table, and initiates the transfer of the drum output page by calling PUTDRQ. After blocking itself and being unblocked by the Drum Interrupt Process, PRINT prints the user's output preceded by a header on the line printer. Thereafter, the print buffer, the user's drum output page, and the user's PSR are released and PRINT returns to the SCHEDULER.



5.10 TIMER INTERRUPT PROCESS

Undebugged user programs have the potential of executing an infinite loop and never relinquishing control of the CPU. To prevent such an occurrence, the SCHEDULER sets the timer to some reasonably large interval every time it gives control to a process. A timer interrupt will occur if the SCHEDULER does not regain control of the CPU within this interval.

No system process should execute excessively long, and therefore the Timer Interrupt Process will PUNT when a system process is interrupted by the timer. An interrupted user process is deactivated, marked with the abnormal termination bit QBIT, and scheduled for termination. After activating the TERMINATOR, the Timer Interrupt Process returns to the SCHEDULER.

Code for the Timer Interrupt Process:

```
*****
*   TIMER INTERRUPT PROCESS   *
*****
TI$ENTRY:  XR LOAD      (LC$PIT);
           IF <0 GO TO  (PSR$MD.X, TI$USER);
           PUNT        ('40);
TI$USER:   AND          (PSR$ST.X, CABIT);
           OR           (PSR$ST.X, QBIT);
           OR           (PSR$ST.X, TBIT);
           XR LOAD      (ADPSRT);
           OR           (PSR$ST.X, ABIT);
           XR LOAD      (LC$SCH);
           SWITCH PROC  (TI$ENTRY);
*****
```

A UNIFYING APPROACH TO SCHEDULING

M. Ruschitzka

Department of Computer Science
Rutgers University
New Brunswick, New Jersey

A Unifying Approach to Scheduling

Manfred Ruschitzka
Rutgers University
R.S. Fabry
University of California

This paper presents a scheme for classifying scheduling algorithms based on an abstract model of a scheduling system which formalizes the notion of priority. Various classes of scheduling algorithms are defined and related to existing algorithms. A criterion for the implementation efficiency of an algorithm is developed and results in the definition of time-invariant algorithms, which include most of the commonly implemented ones. For time-invariant algorithms, the dependence of processing rates on priorities is derived. The abstract model provides a framework for implementing flexible schedulers in real operating systems. The policy-driven scheduler of Bernstein and Sharp is discussed as an example of such an implementation.

Key Words and Phrases: scheduling algorithms, scheduling models, priority, operating systems, processor sharing, implementation efficiency

CR Categories: 4.31, 4.32, 4.34, 4.35, 8.1

Copyright © 1977, Association for Computing Machinery, Inc. General permission is granted to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under grant DAHCIS-73-G6. Authors' addresses: M. Ruschitzka, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903; R.S. Fabry, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

Introduction

The grouping of scheduling algorithms according to common features and parameters [4, 10, 11, 16] has resulted in the definitions of classes of algorithms which aid in analyzing the resulting system behaviors. Owing to their extended set of parameters, however, more sophisticated algorithms which concern themselves with system load [15], job delay [2], deadlock situations [5], working sets [8], and so on are beyond the scope of those classes. In this paper, a classification scheme is suggested which is applicable to arbitrary algorithms. This scheme is based on a model of a generalized scheduling system which manages the resources in a single multiserver system. Contemporary general purpose systems and computer networks which involve the management of a number of different system resources can be modeled as a set of interacting multiserver systems [1]. Owing to its suitability for classifying algorithms, the model leads to the definition of novel classes of algorithms which are related to existing schemes. Furthermore it provides a framework for comparing and evaluating different algorithms in queueing theoretical terms [17], by means of simulations, and in real operating systems. In an implementation, the overhead of the generalized scheduling system is a function of the particular algorithm used. A criterion for implementation efficiency is suggested, and the class of algorithms satisfying this criterion is defined.

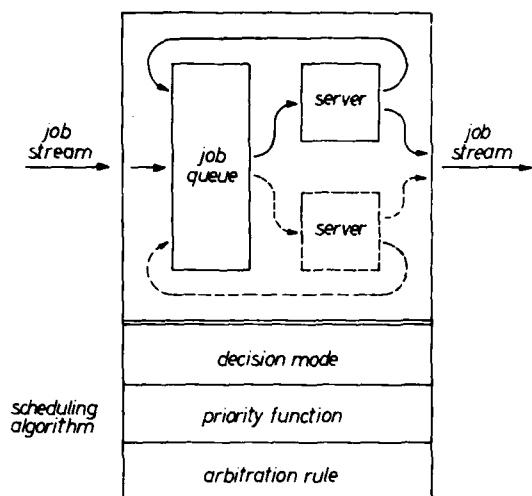
Universal Scheduling System

A *universal scheduling system* (USS) is a generalized scheduler supporting the execution of arbitrary scheduling algorithms for a job stream arriving at a multiserver system. The characteristics, or states, of resident jobs are represented by records which are maintained by the USS. The arbitrary scheduling algorithm may vary in time and is specified in terms of

- a decision mode,
- a priority function, and
- an arbitration rule.

Figure 1 illustrates the structure of a USS. At certain instants in time which are specified by the decision mode, the USS evaluates the priority function for all jobs in the system. The job(s) with the highest priorities is (are) given control of the servers; the arbitration rule is applied in case there are multiple jobs with the same priority. The USS is said to *emulate* an arbitrary scheduling algorithm in the sense that it makes exactly the same scheduling decisions at exactly the same times.

Fig. 1. Structure of a universal scheduling system.



The *decision mode* characterizes the instants in time, or *decision epochs*, at which job priorities are computed and compared and at which one or more jobs are selected for service. The set of jobs being serviced cannot change between two consecutive decision epochs. Depending on the decision mode, algorithms may be

- nonpreemptive,
- quantum-oriented,
- preemptive, or
- processor-sharing.

In nonpreemptive algorithms, jobs are allowed to run to completion; scheduling decisions are only made when a job departs or when an arriving job finds the system empty. In quantum-oriented algorithms, decisions are also made upon completion of a quantum. For infinite quantum sizes, this mode degenerates into the nonpreemptive mode. In preemptive algorithms, a decision is also made upon the arrival of a job. In other words, the decision epochs of preemptive algorithms are the set of all arrival and departure times as well as the quantum completion times. Finally, processor-sharing schemes [12] may be obtained from quantum-oriented ones by letting the quantum size approach zero.

So far, only a few schemes, like round robin and feedback [6, 18], have been studied in processor-sharing mode. Theoretical results about processor-sharing algorithms serve as a good approximation for schemes with small quantum sizes. The decision modes are listed in increasing order of generality in the sense that the set of decision epochs of each mode is a superset of the decision epochs of the previous modes. While these four modes seem to suffice to characterize all interesting algorithms, additional modes are not ruled out. At any rate, any algorithm can be emulated in processor-sharing mode, since it allows decisions to be made continuously. However, the priority function for a particular algorithm may vary with the decision mode in

which this algorithm is emulated. In the less general modes, it is sometimes possible to use simpler priority functions because they are evaluated at discrete intervals only. The three less general modes are included in the classification scheme because they allow simple characterizations of many algorithms and their efficient emulation on a USS.

The *priority function* is an arbitrary function of job and system parameters. At any time, a job's priority is defined as the value of the priority function applied to the current values of the parameters. Following Coffman and Kleinrock [4], we concentrate on naming parameters of the priority functions, rather than trying to elaborate on their forms. Some of the parameters on which priorities can be based are

- memory requirement,
- attained service time,
- total service time,
- external priorities,
- timeliness,
- system load.

The memory requirement serves as a major scheduling criterion in batch processing systems. In interactive systems, it is also important since it is a good measure of swapping overhead, but the attained service time is usually the most important parameter. Some systems assume that the total service time of a job is known in advance. External priorities may be used to differentiate between various classes of user jobs. Timeliness takes into account the fact that the urgency of completing a job may vary in time. The priority may increase in time [2, 13], or, as in deadline scheduling, it may decrease. Greenberger's cost accrual algorithm represents timeliness in its most general form [10]; the goal is the minimization of the accrued cost due to the delay of all jobs in the system. System load is another important parameter owing to its adverse effect on system response. Under heavy load, some schedulers attempt to maintain good response to high priority jobs by discriminating more strongly according to external priorities [15]. Others concentrate on reducing swapping overhead by varying quantum sizes [7]. As opposed to the other parameters we have listed, system load is not a job characteristic; its value is the same for all jobs in the system.

The *arbitration rule* resolves conflicts among jobs with equal highest priority. Usually a first in, first out, or *FIFO*, rule is adopted. Note, however, that the arbitration rule can make the difference between a *FIFO* and a last in, first out, or *LIFO*, policy. In the quantum-oriented mode, tied jobs are often allocated quanta in a *cyclic* manner. In the processor-sharing mode, all jobs with the highest priority are served simultaneously; the arbitration rule is irrelevant. The arbitration rule is therefore not essential for the specification of an algorithm. As with the decision mode, the advantage of specifying an arbitration rule is that it simplifies the priority function.

Priorities and Policy Functions

In the most general case, the priority of a job may be an arbitrary function of an arbitrary number of parameters, including its attained processing time a , the real time r which the job has spent in the computer system, its processing requirement t , an externally assigned importance factor i , some measure of its memory requirement m , and so on:

$$P = P(a, r, t, i, m, \dots). \quad (1)$$

Together with a decision mode and an arbitration rule, this priority function P defines a scheduling algorithm. Of course any transforms of the priority function which preserve equalities or inequalities emulate the same scheduling scheme. Such transforms are said to generate *equivalent* priority functions. Table I lists priority functions, decision modes, and arbitration rules for a few scheduling algorithms.

A large class of important scheduling algorithms can be defined by a priority function of only three arguments: $P = P(a, r, t)$. Algorithms in this class include shortest job first, shortest remaining service time first, longest job first, and longest remaining service time first. A subset of this class is independent of the service time requirement t and can be characterized by a priority function of only two parameters: $P = P(a, r)$. Algorithms in this class are called *unbiased algorithms* because they have no advance knowledge of the job's total service requirement t . Unbiased algorithms include FIFO, service in random order, LIFO, round robin, feedback [6], and some cost accrual schemes [10]. Unbiased algorithms are widely used in real systems, and many of their properties have been investigated. In particular, Kleinrock, Muntz, and Hsu derived tight bounds and a conservation law [14] for exactly this class of algorithms.

An algorithm is called *time-invariant* if the difference between the priorities of two jobs does not change as long as neither of them receives service. Time-invariant algorithms are particularly efficient to emulate.

Table I. Constants: c_1, c_2 ; scheduling parameters: m (memory requirement), r (real time in system), a (attained service time), t (total service time); decision modes: np (nonpreemptive), qo (quantum-oriented), p (preemptive), ps (processor-sharing).

Scheduling algorithm	Priority function	Decision mode	Arbitration rule
smallest memory requirement first	$c_1/m + c_2$, $-m$, etc.	np	arbitrary
FIFO	r	np	arbitrary
LIFO	$-r$	np	arbitrary
round robin	0	qo	cyclic
feedback	$-a$	qo	FIFO
preemptive shortest job first	$-t$	p	FIFO
processor-sharing, longest remaining service first	$t - a$	ps	not applicable

For algorithms which are both time-invariant and unbiased, we conclude

$$P(a, r + x) - P(a, r) = v(x) \quad (2)$$

where $v(x)$ is a function only of x since the difference must not depend on either a or r . Equation (2) is a special case of the Hamel equation [9]. If $P(a, r)$ is bounded in any interval, the unique solution is

$$P(a, r) = Cr + f(a) \quad (3)$$

where C is a constant and $f(a)$ is an arbitrary function in a . Note that $P(a, m, i, r, \dots) = Cr + f(a, m, i, \dots)$ denotes a more general class of time-invariant algorithms.

The constant C in eq. (3) plays an important role. For $C < 0$, a job's priority decreases in real time. Such a policy might be used for jobs whose quick completion is important but whose timeliness decays in real time. Deadline scheduling, LIFO, and some cost accrual schemes are examples of such algorithms. For $C = 0$, priority is a function only of the attained service time. This is the case for feedback schemes which use a FIFO arbitration rule (Table I). If $f(a)$ is also constant, the operation of the USS is determined by the arbitration rule. For example, a quantum-oriented mode and a cyclic arbitration rule yield the round robin algorithm. In the processor-sharing mode, the arbitration rule has no effect and a constant $f(a)$ results in the processor-sharing round robin algorithm. A positive constant C assures that a job's delay is given special consideration. FIFO, the policy-driven scheduler [2], and some of the cost accrual policies belong to this class. As indicated above, feedback schemes may be specified by a priority function with $C = 0$ and a FIFO arbitration rule which serves to determine the priorities of jobs with the same attained service time. Instead of a FIFO arbitration rule, the real-time parameter r may be used to serve the same purpose. In this case, a more complex priority function with $C > 0$ results. This alternative priority function which does not require a FIFO arbitration rule will be discussed under "Examples of Emulations."

If C is nonzero, eq. (3) may be divided by C without altering the emulated algorithm since equalities and inequalities are preserved. We assume in the remainder of this paper that priorities are increasing in real time ($C > 0$). Analogous results can be obtained with $C < 0$. Thus eq. (3) reduces to:

$$P(a, r) = r + F(a), \quad (4)$$

where the arbitrary function $F(a)$ is called the *policy function*¹ of the unbiased time-invariant priority $P(a, r)$. In general, time-invariant priorities are characterized by a policy function F of an arbitrary number of arguments (e.g. attained CPU time, working set size, externally assigned user class, attained channel time, calls to

¹ $F(a)$ is named after the policy function $F(r)$ in Bernstein and Sharp's policy-driven scheduler [6]. Note, however, that $F(a)$ actually corresponds to the inverse of $f(r)$. Physical interpretations of $F(a)$ and its derivative will be presented.

the operating system, etc.). Note that the dimension of the policy function in eq. (4) is real time. We can therefore plot both the position of a job and the policy function in the same real-time / service-time diagram.

Figure 2 shows the relation between the policy function and priorities for two jobs in the system. In this diagram, the priority of a job is given by its vertical distance from the policy function. Independent of their values of r and a , two or more jobs with the same vertical distance from the policy function will therefore have the same priority. If the positions of a group of jobs were plotted at some instant, they would lie on some translation of the policy function in positive or negative direction of real time if and only if all jobs in this group have the same priority. Jobs with different priorities will be positioned on different translations of the policy function, and no job can be positioned above the translated policy function which carries the group of jobs (possibly just one) with the highest priority. Jobs below move upward at unit rate since they receive no service. They will receive service as soon as they catch up with the highest priority group.

For the simple policy function $F(a) = \text{constant}$, the priority of a job increases linearly with the time it spends in the system. Independent of the decision mode, the scheduling system will therefore service jobs to completion in the order of their arrival; this is the FIFO algorithm. Similarly any monotonic decreasing policy function also yields the FIFO algorithm since for such policy functions the term $-F(a)$ in eq. (4) increases while a job is being serviced. Thus no other job can ever reach the priority of the running job.

Equivalent Policy Functions

From the example of the FIFO algorithm, it is apparent that policy functions do not map one-to-one into algorithms. Rather it has been argued that all monotonic decreasing policy functions map into the same algorithm. In general an arbitrary policy function can be replaced by a unique equivalent policy function.

Consider the case of a policy function with a local maximum as depicted in Figure 3 and assume the decision mode of processor sharing. Suppose that after R seconds in the system a test job has reached m seconds of service, where m denotes the local maximum of the policy function. Suppose also that the test job is currently being serviced, i.e. it is in the highest priority group, and that the attained service times of all other jobs in this group are outside the range $[m, b]$, where $F(b) = F(m)$. The priority of all jobs in the highest-priority group at this instant is $P(m, R) = R - F(m)$. Owing to the decreasing values of the policy function above m , the test job gains priority faster than the other jobs and seizes the server until it reaches b seconds of service. At that point, its priority is again the same as that of the other jobs which have previously been in the

highest priority group, namely $R + (b - m) - F(b)$ or $R + (b - m) - F(m)$ or $P(m, R + (b - m))$. Thus, after reaching b seconds of service, the test job will again share the facility with the other highest priority jobs. But exactly the same scheduling sequence would have been achieved if the valley of the policy function $F(a)$ between m and b had been replaced by a horizontal line. This is due to the fact that all highest priority jobs remain on the same translation of the policy function while the test job is being serviced over the horizontal portion. Note that the equality of priorities would be disturbed if any other job in the highest priority group were serviced.

Consider next the case of a policy function with a countable number of local maxima as illustrated in Figure 4. Assume also that a number of highest priority jobs have attained values of the service time which correspond to local maxima of $F(a)$. In this case, different scheduling sequences may result for different shapes of the policy function between the local maxima. For continuously distributed interarrival times, the probability that two jobs reside on local maxima at the same time is zero and such a possibility will be ignored. Thus, assuming continuously distributed interarrival times, an arbitrary policy function may be replaced by an equivalent monotonic increasing policy function, which emulates the same algorithm. It can be shown that this result is true for arbitrary arrival processes.² A unique *normalized* policy function can be obtained from this equivalent monotonic increasing one by adding a constant such that $F(0) = 0$. In the real-time / service-time diagram, the priority of all jobs residing on the normalized policy function is therefore equal to zero.

While normalization by adding a constant to a policy function will always preserve the scheduling sequences of jobs passing through the system, a word of explanation is in order about the scope of the equivalence of monotonic increasing policy functions. This equivalence was shown to be valid for jobs which are continuously serviced in the highest priority group. If a given policy function assures that all jobs which have joined the highest priority group will remain in the highest priority group until they depart, it follows that the equivalent monotonic increasing policy function will result in identical scheduling sequences for all jobs. On the other hand, if the form of a given policy function permits the priority of the highest priority group to assume values less than $F(0)$, the priority of an arriving job, then the replacement of a valley of this function by a horizontal line may decrease a job's priority below $F(0)$. This may lead to preemption of the highest priority group by the new arrival and thus change the scheduling sequence with respect to the new arrival. But since a job cannot attain more service than real time in

² This derivation involves the higher order derivatives of $F(a)$. Since it is beyond the scope of this paper, it will not be presented here.

Fig. 2. Priorities derived from a policy function: $P(a, r) = r - F(a)$.

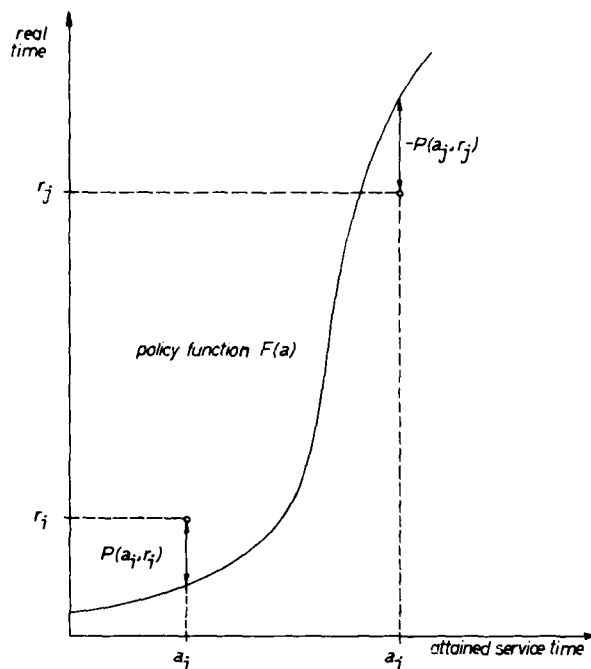


Fig. 3. Policy function with a local maximum and its equivalent.

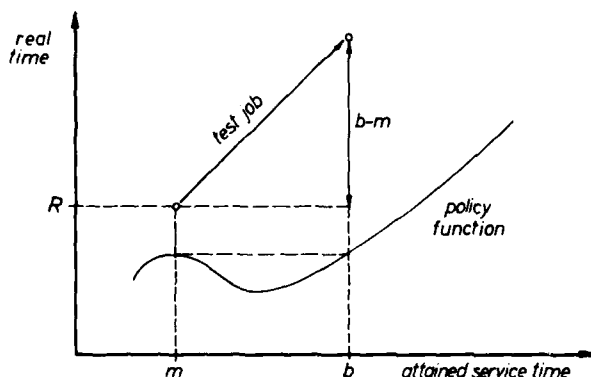
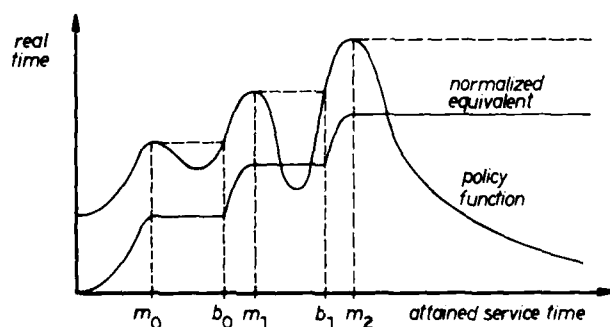


Fig. 4. Policy function and its normalized equivalent.



the system ($a \leq r$), its priority, defined in eq. (4), cannot be less than the priority $F(0)$ of an arriving job unless $F(a) > F(0) + a$ for some range of a . Consequently the replacement of an arbitrary policy function $F(a)$ by its equivalent monotonic increasing one will preserve the scheduling sequences of all jobs passing through the system if all local maxima $F(m_i)$ satisfy $F(m_i) \leq F(0) + m_i$ (cf. Figure 4). Otherwise preemption is possible and only the relative scheduling sequences of preempted highest priority groups are preserved.

Processor Sharing at Different Rates

For the processor-sharing case, the shape of the normalized policy function lends itself to an interpretation of its physical meaning. As outlined in the previous section, a zero slope may allow a job to seize the processor. Conversely a very steep slope causes a job to lose priority quickly. In general, the processing rate of a job is closely related to the derivative of the policy function at the point corresponding to the job's attained service time.

This result can be demonstrated as follows. Assume that at time r the USS services n jobs with the same highest priority P simultaneously on s servers and that n is greater than or equal to s . Assume also that job i ($i = 1, 2, \dots, n$) has been in the system for r_i seconds and has attained a_i seconds of service. Then

$$P = r_i - F(a_i), \quad i = 1, 2, \dots, n. \quad (5)$$

After an infinitesimal time interval Δr , each of the n jobs has gained Δa_i seconds of service, and the priority of the n jobs has changed to

$$P + \Delta P = r_i + \Delta r - F(a_i + \Delta a_i), \quad (6)$$

where

$$s\Delta r = \sum_{i=1}^n \Delta a_i. \quad (7)$$

A Taylor series expansion of F , cancellation of the term P , dividing by Δr , and taking the limit yields an expression for the fraction of real time each job gained in service:

$$\begin{aligned} \lim_{\Delta r \rightarrow 0} \frac{\Delta a_i}{\Delta r} &= \lim_{\Delta r \rightarrow 0} \frac{1 - (\Delta P / \Delta r)}{F'(a_i)} = \frac{s / \sum_{i=1}^n (1 / F'(a_i))}{F'(a_i)} \\ &= \frac{K}{F'(a_i)}, \quad i = 1, 2, \dots, n, \quad K \geq 0, \end{aligned} \quad (8)$$

where K is a proportionality factor which depends on the number of servers, the policy function, the number of jobs in the highest priority group, and their attained service times. Equation (8) states that the service rates of a job in the highest priority group is inversely proportional to the derivative of the policy function evaluated for its attained service. This result is the microscopic equivalent of the well-known fact that the aver-

age service rate of a job is identical to the inverse of the derivative of the *response function* (the average amount of real time a job spends in the system as a function of its service time) [14, 17].

The service rate involves the factor K , which can be interpreted with respect to the dynamic upward and downward motion of the translation of the policy function carrying the highest priority jobs. Since the distance of this translation from the normalized policy function is identical to the priority of the jobs it carries, the rate at which this translation moves is given by the quantity $\lim(\Delta P/\Delta r) = 1 - K$. Jobs with the same priority are effectively in a group, and no job will leave the group until it is completed. While the priority of the highest priority group changes at a rate of $1 - K$, the priorities of all other groups change at unit rate since no service is attained. A lower priority group may therefore merge with the highest priority group. On the other hand, the highest priority group may be preempted by a group of newly arriving jobs with higher priority. From eq. (8) it can be seen that not all jobs in the highest priority group need to receive service simultaneously. First, if the derivative $F'(a)$ is zero in some region, one or more jobs may seize the processor(s). Second, if the derivative $F'(a)$ is infinite for some value of the attained service, one or more jobs may relinquish the processor(s). On the other hand, with a strictly monotonic increasing policy function with finite derivatives, all jobs which have ever received service simultaneously will always be serviced simultaneously.

The special case of a linear policy function whose slope approaches infinity deserves some attention. Under such an algorithm, jobs lose priority at a rate approaching infinity as soon as they receive service. Their priority increase due to the time they spend in the system becomes negligible. Thus the jobs with the least amount of attained service have the highest priority. This is the strategy of the processor-sharing feedback (FB) scheme [6]. In general any "vertical" policy function which is the limit of a monotonic increasing policy function will emulate the processor-sharing FB algorithm.

In the general case, the policy function is a function of n arguments (service time, memory requirement, operating system calls, channel time, etc.), and its derivative is a weighted sum of n partial derivatives. The display of such a policy function requires an $(n + 1)$ -dimensional space. An interesting case arises when the policy function is a function of a linear combination of its parameters: $F(c_1a_1 + c_2a_2 + \dots + c_ra_r)$. If a "generalized attained service" a is substituted for this combination of weighted parameters, the results about normalized policy functions and processor sharing at different rates remain valid. The policy function can be displayed as in Figure 2, the sum of partial derivatives degenerates into a total derivative with respect to the generalized service, and the relative service rates re-

main inversely proportional to this derivative. At least two real systems utilize the notion of such a generalized service in their schedulers; in the policy-driven scheduler on the GE 635 of the General Electric Research and Development Center [2], it is measured in terms of "resource units," and the System Resources Manager in IBM's VS2/2 expresses the service rate in terms of "service units" per second [15].

Implementation

A prominent feature of the USS is its suitability for both theoretical and experimental approaches. Implementing a scheduler as a USS allows algorithms to be changed easily over a wide range. Moreover, such changes may be instigated either externally or internally. Of course, it is often unrealistic to emulate a processor-sharing algorithm because of excessive overhead. But even for algorithms which do not employ processor sharing, the issue of overhead arises in the context of computing and comparing the priorities of all jobs at every decision epoch.

The importance of time-invariant priorities becomes apparent in this context. While any time-invariant algorithm can be implemented in the way suggested below, we shall continue to use the example of unbiased time-invariant algorithms as an illustration. Equation (4) shows that for such algorithms priorities change linearly in real time unless a job attains service. Determining the real time r which a job has spent in the system from its time of arrival e and the current real time c ,

$$r = c - e, \quad (9)$$

and subtracting the current time c from all job priorities (which preserves equalities and inequalities) changes eq. (4) to

$$P(a, c - e) - c = -e - F(a). \quad (10)$$

The priority measure in eq. (10) is particularly suited to an efficient implementation. Since this measure is independent of real time, job entries can be ordered in a queue according to decreasing priorities. At a decision epoch, the USS therefore need not compute the priorities of all jobs but simply picks the job(s) at the head of the queue for execution. The priorities of the preempted jobs are recomputed and their entries are inserted into the queue according to these new values. Essentially this is the scheme implemented in the policy-driven scheduler [2]. There the policy function is a function of the user class and of a linear combination of attained services measured in resource units. The queue is ordered in increasing order of the negative priority measure of eq. (10), and special rules have been introduced to govern the swapping activities. The original implementation of this scheduler was not time-invariant and required an excessive amount of

Table II. Decision modes: np (nonpreemptive), qo (quantum-oriented), pr (preemptive), ps (processor-sharing); scheduling parameters: r (real time in system), i (user class), R (generalized attained service), a (attained service), q (quantum size).

Scheduling algorithm	Priority function	Policy function	Decision mode	Arbitration rule
FIFO	r	0	np	random
policy-driven scheduler[2]	$r - f^{-1}(i, R)$	$f^{-1}(i, R)$	qo	FIFO
two-level preemptive feedback	r , for $a \leq q$, $r - \infty$, for $a > q$	0, for $a \leq q$, ∞ , for $a > q$	pr	FIFO
processor-sharing feedback	$r - \lim_{k \rightarrow \infty} ka$	$\lim_{k \rightarrow \infty} ka$	ps	not applicable

computation for its execution. A minor change in the algorithm, which converted it into a time-invariant one, resulted in a significant improvement of system behavior.

Examples of Emulations

Many algorithms are based on the notion of job classes. Job classes serve to identify jobs with related priorities and can be defined in terms of arbitrary job parameters. Depending on the definition, a job may or may not change class membership during its lifetime in the system. If the priorities of jobs in one class are always to be higher (or lower) than the ones in another class, independent of how long they have been in the system, the priorities of the two classes on a SSS must differ by an infinite amount. Conventional schedulers typically maintain a separate queue for each class. The most common example for these algorithms is the group of n -level feedback algorithms [6], where class membership is based on the attained service. This group can be emulated by using the policy function

$$F(a) = \lim_{z \rightarrow \infty} iz, \text{ for } q_i \leq a < q_{i+1};$$

where $i = 0, 1, \dots, n-1$; $q_0 = 0, q_n = \infty$. (11)

Theoretically the limit in this equation is necessary to guarantee the correct emulation when the real time of a job in the system approaches infinity. For all practical purposes, however, this limit can be approximated by using a large positive integer for z . Jobs which have acquired fewer than q_1 seconds of service are treated in a FIFO fashion. When a job has received q_1 seconds of service, its priority jumps to $r - z$, thus effectively preventing any service allocation as long as there are jobs with fewer than q_1 seconds of service in the system. After the i th quantum q_i , priority is reduced to $r - iz$, delaying service until all jobs have received i quanta.

The multilevel feedback algorithms [14] represent a generalization of the n -level feedback algorithms. While the latter treat all jobs on one level in FIFO fashion, multilevel feedback algorithms allow for the

specification of either FIFO, processor-sharing round robin, or processor-sharing FB for every level. As for the n -level feedback scheme, the job class on level i may be assigned a range of priority numbers such that $-iz \leq P < -(i-1)z$. The priority function $P(a, r)$ is, of course, a generalization of the priority function for n -level schemes:

$$P(a, r) = \lim_{z \rightarrow \infty} \begin{cases} r - zi, & \text{for FIFO,} \\ -zi, & \text{for psRR,} \\ r - z[i - (1/2)(1 - (a - q_i)/(q_{i+1} - q_i))], & \text{for psFB,} \end{cases} \quad (12)$$

where $q_i \leq a < q_{i+1}$;

and $i = 0, 1, \dots, n-1$; $q_0 = 0$; $q_n = \infty$.

If only FIFO and/or processor-sharing FB are specified for the n levels, the policy function $F(a)$ can be determined directly from the relation $P(a, r) = r - F(a)$. If processor-sharing round robin is also specified for one or more levels, the constant C in eq. (3) is zero and eq. (12) cannot uniformly be divided by C to yield $F(a)$. The priority measure of eq. (10) remains valid, however, if the time of arrival e is defined to be a constant for all job classes on processor-sharing round robin levels.

The policy-driven scheduler [2] deserves special credit for proving the feasibility of implementing a scheduler based on a functional parameter—the policy function—in a production oriented system. In this installation, the policy function $f^{-1}(i, R)$ is a function of the user class i and of “resource units” R , a linear combination of various attained services. The specifications for this scheduler as well as for a number of other time-invariant algorithms are summarized in Table II.

The selfish round robin (SRR) algorithms, which include FIFO and processor-sharing round robin, are defined in terms of two parameters α and β , or equivalently in terms of two job classes [11]. The priority of jobs in the highest priority group increases at a rate β , while all other jobs gain priority at a rate α , where $0 \leq \beta \leq \alpha$. Since an arriving job gains priority at a higher rate, it will eventually catch up with the highest priority group, say after a waiting time W . Thereafter it will share the facility equally with the other highest priority jobs. SRR algorithms are emulated by the priority function

$$P(r, W) = \begin{cases} \alpha r, & \text{for } r \leq W, \\ \alpha W + \beta(r - W), & \text{for } r > W, \end{cases} \quad (13)$$

or, after dividing by α ,

$$P(r, W) = \begin{cases} r, & \text{for } r \leq W, \\ (\beta/\alpha)r - W(\beta/\alpha - 1), & \text{for } r > W. \end{cases} \quad (14)$$

Note that $P(r, W) = r$ if $\beta = \alpha$; this priority function specifies the FIFO algorithm (policy function $F(a) = 0$). If $\beta = 0$, the first job in a busy period will retain a zero priority because its waiting time W is zero. Since its

priority is also the highest priority in the system, however, all new arrivals with priority $P(r = 0, W) = 0$ will immediately join the highest priority group. Thus a SRR algorithm with $\beta = 0$ specifies a zero priority function which emulates the processor-sharing round robin algorithm.

Chua and Bernstein [3] have analyzed a parameterized model which lends itself to an analysis of a class of feedback algorithms. This model is quantum-oriented and makes use of a queue with numbered positions. Position 1 contains the job being serviced. After receiving its i th quantum, a job is fed back into queue position π_i , where the set of π_i 's ($i = 0, 1, 2, \dots$) specifies a particular algorithm. Special rules govern collisions with new arrivals and ensure that all jobs in the system are placed in contiguous positions at the top of the queue. After a quantum, the new position of any job can be defined as a function of its previous position, the set of π_i 's, the number of jobs in the system, whether there is a new arrival or not, and the number of attained quanta as well as the total service requirement of the job leaving position 1. Since the priority of a job is monotonic decreasing with its queue position, any monotonic decreasing function of the function for the new position will serve as the priority function emulating the algorithm specified by the set of π_i 's. In the general case, such an algorithm will not be time-invariant since jobs being fed back may be inserted between two previously contiguous jobs in the queue.

The derivation of the policy function $F(a)$ for time-invariant unbiased algorithms was based on the assumption of a positive constant C in eq. (3). For algorithms which cause a job's priority to decrease in real time, however, this constant must be negative. The class of time-invariant unbiased algorithms actually consists of three subclasses which are characterized by $C = 0$, $C > 0$, and $C < 0$.

The latter two subclasses contain the same number of algorithms and, moreover, form a duality. For any algorithm with $C < 0$, the priority function (cf. eq. (4)) can be expressed as

$$P(a, r) = -(r - G(a)).$$

In the real-time / service-time diagram, the priority is still proportional to the vertical distance of a job from the "policy function" $G(a)$, but with opposite sign. Thus the highest priority jobs are positioned on the lowest translation of $G(a)$, and all other jobs must necessarily reside above this translation. Normalization of an arbitrary $G(a)$ results in a monotonic decreasing function, and the derivative of a normalized $G(a)$ is inversely proportional (but with opposite sign) to the service rate of a job in the highest priority group. The term

$$P(a, c - e) + c = e + G(a)$$

may be used as an efficient priority measure in imple-

mentations (cf. eq. (10)). To summarize, for every algorithm $P(a, r) = r - F(a)$ in subclass $C > 0$ there exists a dual algorithm $P(a, r) = -(r - G(a))$ in subclass $C < 0$, where $G(a) = -F(a)$ and dual physical interpretations hold. In contemporary computer systems, scheduling algorithms with $C < 0$ are not common, but LIFO is an important algorithm for many applications in operations research. Clearly LIFO is the dual algorithm to FIFO, since it can be emulated with $G(a) = 0$ or $P(a, r) = -r$.

Summary

The model of a USS provides a unifying mechanism for dealing with arbitrary scheduling algorithms. Some algorithms define priorities in terms of queue structures, while others base their decisions on priority numbers. In formalizing the notion of job priority, the USS relates these two approaches and provides the basis for comparing and evaluating different algorithms. The model suggests a rather natural classification scheme in terms of decision mode, priority function, and arbitration rule and leads to the definition of various classes, including the unbiased and the time-invariant scheduling algorithms. Algorithms which are not time-invariant can be specified in terms of priority functions. Policy functions can be used to emulate time-invariant algorithms. The derivatives of normalized policy functions are shown to control the relative service rate of a job as a function of its attained service. Furthermore, the duality among time-invariant unbiased algorithms is pointed out.

The USS is more than a theoretical tool, however. It lends itself to an efficient implementation for time-invariant policies. In such an implementation, the overhead occurs at decision epochs and consists of computing priority measures for the preempted jobs and inserting them into an ordered queue. For some simple algorithms, this overhead may be slightly higher than for a conventional scheduler which does not use a function as a priority measure. For more sophisticated algorithms, a USS provides a flexible, efficient, and yet uniform framework for implementation. Arbitrary parameters, including system load, job delay, and so on, may be considered and the scheduling algorithm may be modified while the system runs, either dynamically or by external intervention.

Designed to consider a minimum level of acceptable service specified via a policy function for each user group, the policy-driven scheduler [2] represents an implementation for time-invariant algorithms. The results concerning emulation, equivalence of policy functions, and the relationship between policy functions and processing rates are directly applicable to this scheduler and demonstrate its potential generality.

Received June 1975; revised July 1976

Communications
of
the ACM

July 1977
Volume 20
Number 7

References

1. Baskett, F., and Muntz, R.R. Queuing network models with different classes of customers. Proc. Sixth Annual IEEE Int. Conf., San Francisco, Sept. 1972, pp. 205-209.
2. Bernstein, A.J., and Sharp, J.C. A policy-driven scheduler for a time-sharing system. *Comm. ACM* 14, 2 (Feb. 1971), 74-78.
3. Chua, Y.S., and Bernstein, A.J. Analysis of a feedback scheduler. *SIAM J. Compig.* 3, 3 (Sept. 1974), 159-176.
4. Coffman, E.G. and Kleinrock, L. Computer scheduling methods and their countermeasures. Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Montvale, N.J., pp. 11-21.
5. Coffman, E.G., Elphick, M.J., and Shoshani, A. System deadlocks. *Computing Surveys* 3, 2 (June 1971), 67-78.
6. Coffman, E.G., and Kleinrock, L. Feedback queueing models for time-shared systems. *J. ACM* 15, 4 (Oct. 1968), 549-576.
7. Coffman, E.G. Analysis of two time-sharing algorithms designed for limited swapping. *J. ACM* 15, 3 (July 1968), 341-353.
8. Denning, P.J. The working set model for program behavior. *Comm. ACM* 11, 5 (May 1968), 323-333.
9. Feller, W. *An Introduction to Probability Theory and Its Applications, Vol. I*. Wiley, New York, Third Ed., Rev. Printing, 1970.
10. Greenberger, M. The priority problem and computer time sharing. *Manage. Sci.* 12, 11 (July 1966), 888-906.
11. Kleinrock, L. A continuum of time-sharing scheduling algorithms. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., pp. 453-458.
12. Kleinrock, L. Time-shared systems: A theoretical treatment. *J. ACM* 14, 2 (April 1967), 242-261.
13. Kleinrock, L. A delay dependent queue discipline. *Nav. Res. Log. Quart.* 11, 4 (1964), 329-341.
14. Kleinrock, L., Muntz, R.R., and Hsu, J. Tight bounds on the average response time for time-shared computer systems. Information Processing 71, North-Holland Pub. Co., Amsterdam, pp. 124-133.
15. Lynch, H.W., and Page, J.B. The OS/VS2 release 2 system resources manager. *IBM Systems J.* 13, 4 (1974), 274-291.
16. McKinney, J.M. A survey of analytical time-sharing models. *Computing Surveys* 1, 2 (June 1969), 105-116.
17. Ruschitzka, M. System resource management in a time sharing environment. Ph.D. Th., Dept. of EECS, U. of California, Berkeley, Nov. 1973.
18. Schrage, L.E. The queue M/G/1 with feedback to lower priority queues. *Manage. Sci.* 13, 7 (1967), 466-474.

To appear in the special issue "On Interfaces with Computer Science"
of Operations Research in late 1977 or early 1978.

AN ANALYTICAL TREATMENT
OF POLICY FUNCTION SCHEDULES

M. Ruschitzka

June 1977

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research
Projects Agency of the Department of Defense under Grant #DAH15-73-G6
to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the
author and should not be interpreted as necessarily representing the
official policies, either expressed or implied, of the Advanced
Research Projects Agency or the U. S. Government.

ABSTRACT

This paper presents an analysis of time-sharing computer facilities with scheduling algorithms defined in terms of priority functions. We consider the class of algorithms in which a job's priority is defined by the difference between the time it spent in the system and an arbitrary function F of its attained service, where F is called the policy function.

Our main result, the average response time for a job conditioned on its service requirement, applies to a broad class of policy functions. The derivation is based on the model of a processor-sharing M/G/1 queuing system and does not use transforms. A method involving the decomposition of a non-Markovian process is shown to simplify the analysis.

Properties of the average response time resulting from policy function schedulers are discussed and related to those of other known time-sharing scheduling algorithms. For the examples of linear, exponential, and composite policy functions we plot the response times. The flexibility and the limitations of policy functions with respect to the discriminatory treatment of short jobs versus long ones are demonstrated, and the optimal selection of a policy function with respect to a given overall performance criterion is discussed.

AN ANALYTICAL TREATMENT OF POLICY FUNCTION SCHEDULERS

Bernstein and Sharp [3] designed and implemented the first policy function scheduler in 1969. This scheduler attempted to avoid unnecessary system overhead, in particular swapping, by specifying (and adhering to) a minimal level of acceptable service to users. Parameterized scheduling algorithms have allowed system designers and installation managers to tune systems before, but the use of arbitrary functions, rather than discrete parameters, permits a more flexible specification of the desired system behavior. It is an indication of their usefulness that variations of policy function schedulers have since been adopted for production systems, e.g. IBM's OS/VS2 Release 2 [8], as well as research oriented systems like HYDRA [6]. The implementations of these schedulers have not been without problems. It is tempting to increase the level of sophistication of a policy function to a point where the incurred overhead exceeds the gain in the quality of service. Furthermore, with few theoretical results available, the choices of the forms and of the arguments of policy functions are often based on intuition. Consequently, the process of tuning a system tends to proceed in an ad-hoc fashion and little insight is gained with respect to the intrinsic characteristics of policy function schedulers.

Any scheduling algorithm can be expressed as a priority scheduling algorithm. In the latter, the scheduler evaluates a priority function $P(p_1, p_2, p_3, \dots)$ of an arbitrary number of job parameters p_i (service time, memory requirement, system calls, channel time, external job class, etc.) for all competing jobs and allocates the resource(s) to the job(s) with the highest priority value(s). We shall review some of the general properties of such priority functions. For a more comprehensive treatment as well as a general introduction to priority scheduling algorithms the reader is referred to Ruschitzka and Fabry [10]. It was shown there that the overhead for evaluating the priority function can be considerably reduced if the priority function is of the form

$$P(r, p_2, p_3, \dots) = Cr - f(p_2, p_3, \dots), \quad (1)$$

where the first job parameter, r , is the real time a job has spent in the system, C is a constant, and the arbitrary function f of the remaining job parameters is called the policy function. Note that equation (1) defines the relation between policy functions and the notion of priorities. The choice of the instants in time at which the priority function P is evaluated characterizes a scheduling algorithm as non-preemptive, preemptive, quantum-oriented, processor-sharing, etc.. Processor-sharing algorithms are the most general in that they can be used to simulate all others [10]. Because of this generality and also because of

the appealing simplicity of the analytical results, only processor-sharing algorithms will be considered henceforth.

If the constant C in equation (1) is negative, a job's priority decreases in real time. Such a priority function would be used to implement deadline scheduling or LIFO. If $C=0$, the priority is independent of real time. We are interested in the case $C>0$, i.e. those algorithms in which a job's priority increases with the time it spends in the system. In this case, the term on the right hand side of equation (1) can be divided by C without changing the algorithm, since equalities and inequalities of the priorities of any two jobs are preserved. Furthermore, if the policy function is a function of only the attained service time, a , the priority function specializes to

$$P(r,a) = r - F(a), \quad (2)$$

where the new policy function $F(a)$ is simply $f(a)/C$. In this paper, we consider only the class of algorithms defined by priority functions of the form of equation (2). Different forms of the policy function $F(a)$ define different algorithms, and for a variety of conventional time-sharing algorithms the defining policy functions are known. For example, $F(a)=d$, where d is any constant, specifies the FIFO algorithm since the priority of a job will increase linearly with the time it spent in the system. We briefly summarize two results concerning the equivalence and the normalization of policy functions [10]. Policy functions are called

equivalent if their evaluations result in identical scheduling sequences. Thus, the policy functions $F_1(a)=0$ and $F_2(a)=d$ ($d \neq 0$) are equivalent since both yield the FIFO scheduling sequence. However, any set of equivalent policy functions $\{F\}$ can be represented by a unique normalized policy function which is (weakly) monotonic increasing and satisfies $F(0)=0$ [10]. For FIFO, the normalized policy function is $F(a)=0$. Without loss of generality, policy functions are assumed to be normalized in the sequel.

1. MODEL AND APPROACH

It is our goal to derive the relationship between a given policy function and the resulting system behavior. We shall use the response function, the average time a job spends in the system conditioned on its service requirement, to describe the system behavior in equilibrium. The system is modeled by an M/G/1 queuing system, i.e. a system with Poisson job arrivals, a general (arbitrary) distribution of job service times which are mutually independent, and one server. In addition to processor-sharing mode, we assume that the system is work-conserving, i.e. that preemption and resumption of a job will not require additional service time. In sum, we model a time-sharing computer facility as a work-conserving, processor-sharing M/G/1 queuing system driven by a policy function $F(a)$.

When a job arrives at the system, it adds its service time to the unfinished work, the work (in units of service time) which the system has yet to perform on resident jobs. Between job arrivals, the unfinished work decreases in time at unit rate until it reaches zero. Thereafter, the system is free of jobs, the unfinished work remains zero, and the system is said to be idle until the next job arrives. At this epoch the unfinished work jumps to the value of the next job's service time. It will then decrease linearly until either another job arrives or the job is serviced to completion, whichever occurs first. Periods during which the unfinished work is positive are called busy periods. In equilibrium, every busy period is followed by an idle period and vice versa. For our analysis, we shall study the system over a large number of such periods until equilibrium is reached.

We begin to observe the system at the beginning of an arbitrary busy period at time zero and number jobs in the order of their arrival ($j=0,1,2, \dots$). Their arrival epochs and service time requirements will be denoted by the random variables T_j and S_j respectively ($T_0=0$). I_j stands for the time between the arrivals of jobs $j-1$ and j , i.e. $I_j = T_j - T_{j-1}$ ($j=1,2, \dots$). We express delays and other random variables of interest for a particular (but arbitrary) job, say job number n , and refer to this job as the test job. Introducing special symbols for its arrival epoch $t=T_n$ and its service requirement $x=S_n$, we denote its

residence time in the system (the time between arrival and departure) by $R(t,x)$. This residence time consists of the test job's service time x and its waiting time $W(t,x)$:

$$R(t,x) = W(t,x) + x. \quad (3)$$

It often proves advantageous to relate the waiting time of a job to the amount of service (work) performed on other jobs. With this method, the author [9] obtained considerably simplified derivations of the response functions of such algorithms as the processor-sharing round robin and feedback schemes. Using this approach, we distinguish between two groups of jobs: the early arrivals ($j < n$) which arrive before the test job and the late arrivals ($j > n$) which arrive after it. We define the early work $V_E(t,x)$ (late work $V_L(t,x)$) as the total service performed on all early (late) arrivals during the test job's residence time. Clearly, an early arrival which departed prior to the test job's arrival at time t cannot benefit from the early work. Similarly, a job arriving after the test job's departure cannot contribute to the late work. With these definitions, equation (3) may be rewritten as

$$R(t,x) = V_E(t,x) + V_L(t,x) + x. \quad (4)$$

Note that the residence time, the early work, and the late work are stochastic processes defined for any job n ($n=0,1,2, \dots$) arriving at time $t=T_n$. Equilibrium values are obtained by letting n (and, thus, t) approach infinity.

The response function $R(x)$ can now be expressed as the time average of the residence time in equation (4). With the definition of the time average $Q(x)$ of a stochastic process $Q(t,x)$ with a constant parameter x ,

$$Q(x) = \lim_{T \rightarrow \infty} (1/T) \int_0^T E[Q(t,x)] dt, \quad (5)$$

we obtain

$$R(x) = V_E(x) + V_L(x) + x. \quad (6)$$

Unfortunately, the early and the late work processes are not Markovian. However, a method involving a further decomposition of the early work process will be shown to yield to an analytical treatment for all policy functions of the form $F(a) \leq a$.

The significance of the constraint $F(a) \leq a$ on policy functions deserves some attention. Since the attained service, a , of a job is included in the time it spent in the system, r , we always have $r - a \geq 0$. It follows from equation (2) and $F(a) \leq a$ that a job's priority cannot be negative, since $P(a,r) = r - F(a) \geq r - a \geq 0$. The priority loss due to attaining service always dominates the priority gain due to spending time in the system. A newly arriving job has zero priority and will not be serviced until its priority becomes equal to that of the highest priority job(s) in the system. After joining the highest priority group, a job will remain in the highest priority group until it departs. The policy function scheduler assures this by processing the members of

this group at (possibly) different rates [10]. Note that jobs join the highest priority group in the order of their arrival. Thus, a resident job is either waiting with no service attained, or it is a member of the highest priority group and, thus, being serviced. Just prior to its departure, the test job is being serviced and has the same (highest) priority as all early arrivals which are still resident. Similarly, all resident late arrivals which have attained any service at all also have this same highest priority. These properties play an important role in our derivation. If the policy function is not constrained by $F(a) \leq a$, the priority of the highest priority group may become negative. Thus, a new arrival with zero priority may preempt the highest priority group. Furthermore, the priority of a resident early arrival may be lower than that of the test job when the test job departs. In this paper, we shall adhere to the constraint $F(a) \leq a$.

In terms of notation, λ will denote the job arrival rate. The random variable S and the functions G and g stand for the service time requirement of a job, the service time distribution function and its probability density function respectively. G^c denotes the complement of G : $G^c(x) = 1 - G(x)$. A glossary at the end of the paper summarizes the symbols and notation used in the analysis. It also contains a useful identity for the moments of the truncated service time $S_{\leq x}$, $S_{\leq x} = \min[S, x]$, which is frequently used in the derivation. The analysis does not make use of

transforms and, thus, illustrates the physical behavior of the system in time.

2. DECOMPOSITION OF THE EARLY WORK PROCESS

The early work is the amount of service which early arrivals (jobs $j < n$) attain during the test job's residence. Equivalently, it may be defined as the service which early arrivals attain prior to the test job's departure minus the service they attain prior to its arrival. There are two types of early arrivals: those which are serviced to completion and exit before the test job departs, and those which are still resident at the test job's departure epoch. We consider the latter type first.

As noted in the preceding section, the priority of the test job equals that of the resident early arrivals at the departure epoch of the test job. Thus, with a_j denoting the attained service of a resident job j , and using $L=R(t,x)$ for the residence time of the test job, we have

$$P(L,x) = L - F(x) = L + t - T_j - F(a_j) = P(L + t - T_j, a_j).$$

Note that $T_j < t = T_n$ since $j < n$. Solving for a_j , we obtain

$$a_j = F^{-1}(F(x) + t - T_j), \quad (7)$$

where F^{-1} denotes the inverse of the policy function F .

Equation (7) specifies the attained service a_j of a resident early arrival at the test job's departure epoch. Early arrivals of the other type will already have departed after receiving their service requirement S_j . However, had the service requirement S_j of such a departed early arrival been larger than a_j , this job would still be resident with exactly a_j seconds of attained service. Therefore, any early arrival will have attained the minimum of S_j and a_j respectively when the test job departs. The total amount of the early work, which is to be performed during the test job's residence time, can now be expressed as the sum of the individual minima minus the service performed prior to the test job's arrival at epoch t . The latter is simply the sum of the durations of all busy periods up to epoch t , or $t-H$, where H denotes the sum of the durations of all idle periods preceding t . Thus, we have

$$V_E(t, x) = \sum_{j=0}^{n-1} \min[S_j, F^{-1}(F(x) + t - T_j)] - (t - H). \quad (8)$$

By expressing t and T_j as sums of interarrival times and subtracting S_j from the minima we get

$$V_E(t, x) = \sum_{j=0}^{n-1} [S_j - I_{j+1}] + H - \sum_{j=0}^{n-1} \max[0, S_j - F^{-1}(F(x) + t - T_j)]. \quad (9)$$

In the first two terms of equation (9) we recognize the expression for the unfinished work $U(t)$, also called the virtual delay [11]. It is identical to the actual delay of the test job under FIFO. Naming the third term $V_\Delta(t, x)$,

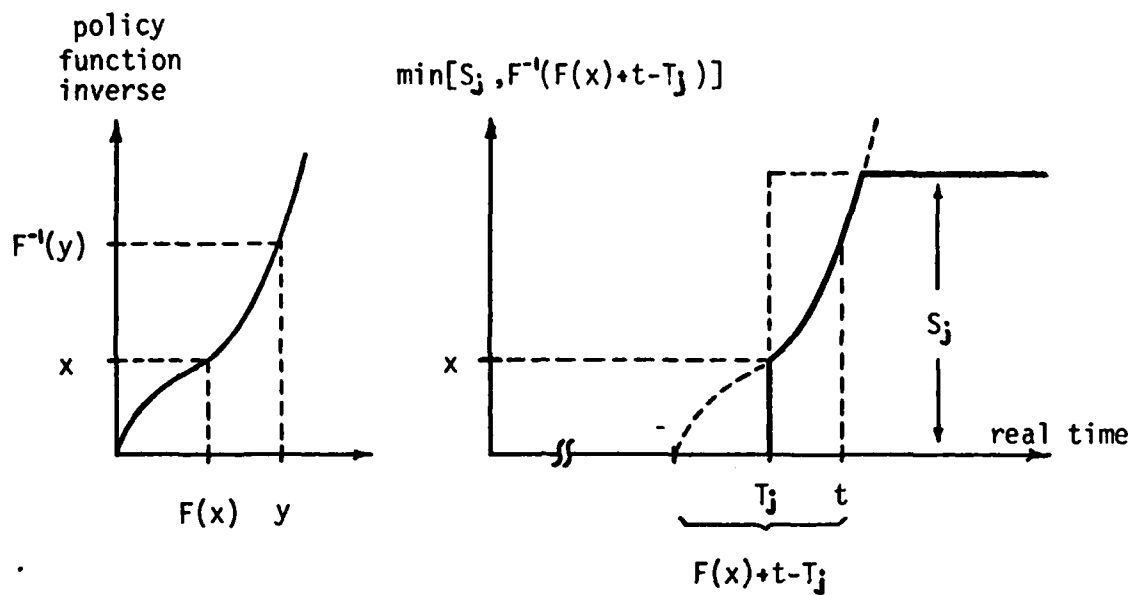
$$V_\Delta(t, x) = \sum_{j=0}^{n-1} \max[0, S_j - F^{-1}(F(x) + t - T_j)], \quad (10)$$

we decompose the early work by rewriting equation (9) as

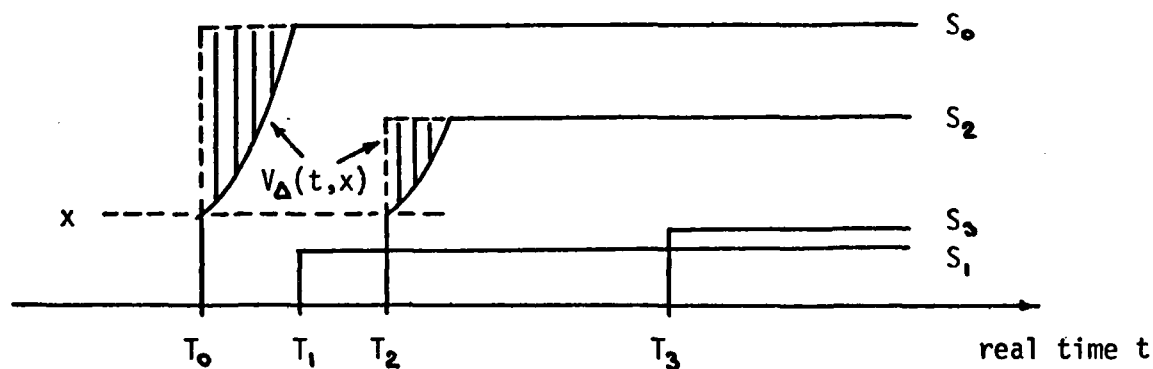
$$V_E(t,x) = U(t) - V_\Delta(t,x). \quad (11)$$

As an aid to intuition, this decomposition is depicted in Figure 1. Assuming some fixed value for the test job's service requirement x , Figure 1.a illustrates how the minimum in equation (8), i.e. the attained service of job j at the departure epoch of the test job, is obtained via the inverse F^{-1} of the policy function. The minima of four jobs forming the first busy period have been superposed in Figure 1.b. The shaded areas represent the maxima in equation (10), i.e. the individual contributions to the process $V_\Delta(t,x)$. Summing up the individual minima and subtracting the work t , which has already been performed, yield the total for the early work. Figure 1.c shows that this total is identical to the unfinished work $U(t)$ minus the work $V_\Delta(t,x)$. A continuation of this diagram over several busy periods consists of probabilistic replicas of the first one separated by idle periods.

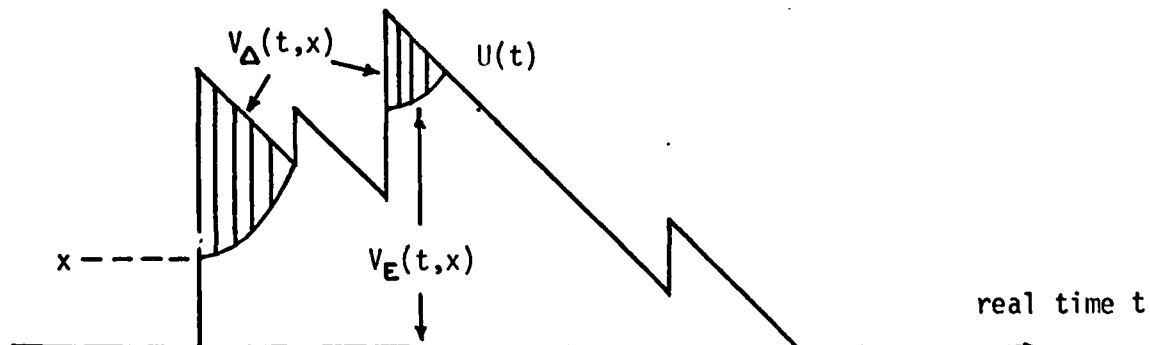
The form of equation (11) suggests that the time average $V_E(x)$ can be obtained as the difference of the time averages U and $V_\Delta(x)$. However, we must first verify that the expression for $V_\Delta(t,x)$ does not cause $V_E(t,x)$ to become negative. In that case, equation (11) would have to be replaced by $V_E(t,x) = \max[0, U(t) - V_\Delta(t,x)]$, since the early work is a nonnegative process with busy periods defined analogously to those of the unfinished work. However, the



(a) Contribution of job j to the early work $V_E(t, x)$.



(b) Superposed contributions to $V_E(t, x)$ and $V_D(t, x)$



(c) The difference between unfinished and early work.

Figure 1. Decomposition of the early work.

already adopted constraint on the policy function, $F(x) \leq x$, also assures the positivity of $V_E(t, x)$ throughout any busy period of $U(t)$. This is no coincidence. Since this constraint assured a non-preemptive mode of operation, a test job which arrives during a busy period and has a non-zero service requirement must always wait for some work to be done on resident early arrivals, i.e. $V_E(t, x) > 0$. We formalize this result in the following theorem.

Theorem: The busy periods of the processes $U(t)$ and $V_E(t, x)$ are identical for all positive values of t and x if and only if $F(x) \leq x$.

Proof: By definition, the early work is a subset of the unfinished work and, thus, a busy period of the early work cannot be longer than the corresponding busy period of the unfinished work. It could be shorter though. On the other hand, equation (11) shows that $V_E(t, x)$ cannot become zero (and terminate its busy period before $U(t)$ does) as long as $U(t) > V_\Delta(t, x)$. The expressions for $U(t)$ and $V_\Delta(t, x)$ are given in equation (9). Since all busy periods are probabilistic replicas of each other, we limit our attention to a single one, say the first one, and set $H=0$. A sufficient condition for $U(t) > V_\Delta(t, x)$ is obtained by requiring the individual j -terms in equation (9) to be positive:

$$S_j - I_{j+1} > S_j - F^{-1}(F(x) + t - T_j), \quad 0 \leq j \leq n-1.$$

After cancelling S_j and expressing $t-T_j$ as a sum of interarrival times, this condition can be rewritten as

$$I_{j+1} < F^{-1}(F(x) + I_{j+1} + I_{j+2} + \dots + I_n).$$

Clearly, the constraint $F(x) \leq x$ assures this sufficient condition for all $x > 0$. If $n=1$, j assumes only the value 0, and the j -term with $j=0$ is identical to $U(t) - V_{\Delta}(t, x)$. This special case shows that the constraint $F(x) \leq x$ is not only a sufficient, but also a necessary condition for the identity of the busy periods of $U(t)$ and $V_E(t, x)$.

With the positivity of equation (11) assured throughout any busy period, the time average of the early work is indeed the difference of the time averages for $U(t)$ and $V_{\Delta}(t, x)$:

$$V_E(x) = U - V_{\Delta}(x). \quad (12)$$

The time average of the unfinished work is well-known (see Wolff [11] for a derivation which does not use transforms),

$$U = \lambda ES^2 / 2(1-\rho)$$

where $\rho = \lambda ES$ denotes the system load, and $V_{\Delta}(x)$ is defined by equation (5):

$$V_{\Delta}(x) = \lim_{T \rightarrow \infty} (1/T) \int_0^T EV_{\Delta}(t, x) dt. \quad (13)$$

For Poisson arrivals, the contributions of the various arrivals to $EV_{\Delta}(t, x)$ are equal to the integral of the expected contribution of a job arriving at epoch t times the

probability of the occurrence of such an arrival. From the definition of $V_{\Delta}(t, x)$ in equation (9) we get

$$EV_{\Delta}(t, x) = \int_0^t \int_{F^{-1}(F(x)+t-\xi)}^{\infty} [s - F^{-1}(F(x)+t-\xi)] g(s) ds \lambda d\xi.$$

Before this expectation can be substituted in equation (13) to obtain the time average, the lower integration boundary for ξ must be adjusted. If the policy function $F(x)$ assumes a finite value for $x \rightarrow \infty$, the inverse $F^{-1}(F(x)+t-\xi)$ is not defined for arguments larger than $F(\infty)$. In that range, the flatness of the policy function prohibits contributions of arrivals which occurred too far in the past. Thus, the lower boundary 0 should be replaced by $\max[0, F(x)+t-F(\infty)]$ to handle the two cases of finite and infinite values of $F(\infty)$. However, by solving the two cases separately, it can be shown that the two boundaries $\max[0, F(x)+t-F(\infty)]$ and $F(x)+t-F(\infty)$ are interchangeable under the limit of equation (13). Choosing the simpler boundary, we have

$$V_{\Delta}(x) = \lim_{T \rightarrow \infty} (\lambda/T) \int_0^T \int_{F(x)+t-F(\infty)}^{\infty} [s - F^{-1}(F(x)+t-\xi)] g(s) ds d\xi dt.$$

Substituting $y = F^{-1}(F(x)+t-\xi)$ and evaluating the innermost integral yield

$$V_{\Delta}(x) = \lim_{T \rightarrow \infty} (\lambda/T) \int_0^T \int_x^{\infty} F'(y) [ES - ES_{<y}] dy dt$$

and the final expression is obtained by taking the limit and integrating by parts:

$$V_{\Delta}(x) = -\lambda F(x)[ES - ES_{<x}] + \lambda \int_x^{\infty} F(y)G^c(y) dy. \quad (14)$$

Rather than deriving this time average from the defining equations, the method of triangular arrays [4] can be adopted. Starting from the definition of $V_{\Delta}(t,x)$ in equation (10), we get

$$V_{\Delta}(x) = \lambda \int_0^{\infty} E[\max[0, S - F^{-1}(F(x) + t)]] dt.$$

Again, the upper integration boundary must be adjusted to handle finite values of $F(\infty)$. We replace it by $F(\infty) - F(x)$ to handle the general case. Substitution of $y = F(x) + t$ yields

$$V_{\Delta}(x) = \lambda \int_x^{\infty} E[\max[0, S - y]] F'(y) dy,$$

and expressing the expectation of the maximum as $ES - ES_{<y}$ followed by an integration by parts results in the expression given by equation (14). This method offers an intuitive interpretation. Each job contributes a certain area (the shaded area in Figure 1) to $V_{\Delta}(t,x)$. The time average $V_{\Delta}(x)$ is simply the expected value of this contribution multiplied by the arrival rate (the number of contributions per unit time).

3. THE LATE WORK PROCESS

The late work is the amount of service which jobs arriving after the test job will attain during the test job's residence time. Together with the early work, it represents the delay encountered by the test job. Recall

that the test job's priority remains non-negative for policy functions of the form $F(x) \leq x$ and that the initial priority of an arriving job is zero. To attain its first service allocation, a late arrival must gain priority by waiting until it catches up with the highest priority group. Thereafter, it will remain in the highest priority group until it departs.

Similar to the approach used to determine the early work, we shall relate job priorities at the departure epoch of the test job in order to determine the late work. Assuming again that the test job n remains in the system for L seconds and that its service requirement is $x=S_n$, we observe a job $j>n$ which arrives i seconds after the test job ($i<L$) and attains A seconds of service by the time the test job departs. Note that a late arrival will attain no service at all unless it catches up with the highest priority group. Equating the priorities of the test job and a resident late arrival,

$$P(L,x) = L-F(x) = L-i-F(A) = P(L-i,A),$$

we obtain the amount of attained service as

$$A = F^{-1}(F(x)-i). \quad (15)$$

For A to be positive, i.e. for the late arrival to catch up with the highest priority group before the test job exits, i must be less than $F(x)$. Thus, no job arriving $F(x)$ seconds after the test job will receive any service during its

residence time L . Note that the constraint $F(x) \leq x$ assures that L is not less than $F(x)$, since the test job's residence time L cannot be shorter than its service requirement x . Consequently, there exists a deterministic upper bound $F(x)$ on the interval during which late arrivals may occur and still get serviced to some degree. This implies that the late work does not depend on the early work which the test job encounters!

Departures of late arrivals are handled like those of early arrivals. If a late arrival departs before the test job, its service requirement S must have been less than the value of A in equation (15). Thus, any late arrival will contribute $\min[S, A]$ to the late work, provided it arrives within $F(x)$ seconds after the test job's arrival. Making use of Poisson arrivals, the expectation of the sum of the contributions of the late arrivals may again be replaced by an integral. The time average for the late work therefore amounts to

$$V_L(x) = \int_0^{F(x)} E[\min[S, F^{-1}(F(x)-i)]] \lambda \, di.$$

Expressing the expectation of the minimum in terms of the expectation of the truncated service time, we have

$$V_L(x) = \lambda \int_0^{F(x)} E S_{<F^{-1}(F(x)-i)} \, di.$$

Substituting y for $F^{-1}(F(x)-i)$ and integrating by parts yield the final expression

$$V_L(x) = \lambda F(x) ES_{<x} - \lambda \int_0^x F(y) G^c(y) dy. \quad (16)$$

It is quite common among scheduling algorithms that the expressions for the early and the late work are more complicated than the response functions themselves. The class of policy functions $F(x) \leq x$ is no exception. We proceed by presenting the overall result.

4. PROPERTIES OF RESPONSE FUNCTIONS

With the time averages for the early and the late work established, we obtain the response function from equation (6) by substituting equations (12), (14), and (16) and using the symbol ρ for the load λES :

$$R(x) = U - \lambda \int_0^\infty F(y) G^c(y) dy + \rho F(x) + x. \quad (17)$$

This result is valid for work-conserving, processor-sharing M/G/1 systems driven by arbitrary policy functions whose normalized equivalents satisfy the constraint $F(x) \leq x$. The average waiting time $W(x)$ of a job with a service requirement of x seconds is equal to $W(x) = R(x) - x$:

$$W(x) = U - \lambda \int_0^\infty F(y) G^c(y) dy + \rho F(x). \quad (18)$$

Note that only the last term is a function of x . The first two terms are constant and have the value $W(0)$. Thus, the shape $W(x) - W(0) = \rho F(x)$ of the average waiting time is independent of the shape of the service time distribution; in fact, it is simply the policy function multiplied by the load ρ . The values of $W(0) = R(0)$ have bounds determined by

the constraint $0 \leq F(x) \leq x$. By evaluating equation (18) for $x=0$ and the policy functions $F(x)=0$ and $F(x)=x$ these bounds can be expressed in terms of the average waiting time U of the FIFO algorithm:

$$\rho U \leq R(0) = W(0) \leq U.$$

The lower bound is obtained with $F(x)=x$ which tends to increase the response of long jobs. Conversely, $F(x)=0$, the FIFO algorithm, maximizes the response for short jobs and favors long jobs.

Using the overall average residence time R as a performance criterion, where R is defined as

$$R = \int_0^{\infty} R(x) g(x) dx,$$

the optimal form of $F(x)$ can be determined as a function of the service time distribution by minimizing R . From equation (17) we obtain

$$R = U + ES - \lambda \left[\int_0^{\infty} F(y) G^c(y) dy - ES \int_0^{\infty} F(x) g(x) dx \right].$$

Noting that only the term having brackets involves the policy function F , we call this term $-\lambda[M]$ and attempt to maximize M . For the policy function $F(y)$ we substitute an integral over its derivative and integrate the second integral in M by parts to get

$$M = \int_0^{\infty} \int_0^y F'(x) dx G^c(y) dy - ES \int_0^{\infty} F'(x) G^c(x) dx.$$

Finally, reversal of the order of integration of x and y

yields the expression to be maximized:

$$M = \int_0^{\infty} F'(x) \left[\int_x^{\infty} G^c(y) dy - ES G^c(x) \right] dx. \quad (19)$$

The bracketed term plays an important role in reliability theory [2]. It is used to characterize the distribution function G of a random variable S with respect to its residual life [5]. In particular, a distribution function G is said to be \mathcal{G} -MRLA (\mathcal{G} -MRLB), i.e. it has its mean residual life bounded above (below), if and only if

$$\int_x^{\infty} G^c(y) dy \begin{matrix} \leq \\ (\geq) \end{matrix} \mathcal{G} G^c(x).$$

The mean residual lives of a large class of distributions are bounded with respect to $\mathcal{G}=ES$. For ES-MRLA distributed service times S , it follows from equation (19) that $F'(x)=0$, i.e. $F(x)=0$ or FIFO, maximizes M and thus minimizes the average overall residence time. This is not surprising since Wolff [12] established the superiority of FIFO over both the round robin and the feedback algorithms for the more general work-conserving $G/G/1$ systems. Consider the case when the service time is not ES-MRLA distributed. Since $F(x)$ need not be continuous, the derivative $F'(x)$ may contain Dirac pulses. Then, the optimal policy function can be obtained from equation (19) via simple optimization techniques [1]. It is not difficult to show that optimal policy functions satisfying the constraint $F(x) \leq x$ consist of sequences of a constant portion, a jump to the identity function, and a portion equal to the identity function. As

an example, the composite policy function in Figure 4 (to be discussed later) illustrates such an optimal policy function consisting of a single sequence of the three portions.

Returning to the expression for the response function in equation (17), we note that it may be viewed as a transformation of policy functions into response functions. The inverse transformation is obtained by a simple differentiation and the application of the boundary condition $F(0)=0$ for normalized policy functions:

$$F(x) = [R(x) - x - R(0)] / \rho.$$

Of course, not any arbitrary response function can be obtained by a policy function. First of all, we are limited by the constraint $F(x) \leq x$ which implies

$$R(x) \leq R(0) + x(1+\rho).$$

But there are at least three more constraints which are generally valid for any scheduling algorithms which base their priorities solely on attained service time and time in system. First, the residence time cannot be smaller than the service time which it includes:

$$R(x) \geq x.$$

Second, an intrinsic constraint on response functions, regardless of the scheduling algorithms used, limits the discriminatory treatment of short jobs versus long jobs. This constraint is referred to as a conservation law

[5, p.197]:

$$\int_0^{\infty} R(x) G^c(x) dx = ES^2/2(1-\rho). \quad (20)$$

Since the integral of the response function weighted by the (decreasing) complementary distribution function must be constant, a favorable treatment of short jobs (small $R(x)$ for small x) implies delays for long jobs and vice versa. It is easily checked that our result in equation (17) satisfies this conservation law. Third, tight upper and lower bounds [5, p.199] have been established on response functions:

$$\frac{\lambda ES_{<x}^2/2}{1-\lambda ES_{<x}} + x \leq R(x) \leq \frac{\lambda ES^2/2}{(1-\rho)(1-\lambda ES_{<x})} + \frac{x}{1-\lambda ES_{<x}}. \quad (21)$$

Applying Little's theorem [7] to an infinitesimally small service time interval dx , the response function $R(x)$ and the average density $n(x)$ of resident jobs with x seconds of attained service, can be related [5, p.164]:

$$n(x) = \lambda R'(x) G^c(x).$$

A substitution of the derivative $R'(x)$ from equation (17) yields

$$n(x) = \lambda [1 + \rho F'(x)] G^c(x) + \lambda R(0) \delta(0), \quad (22)$$

where the Dirac pulse $\delta(0)$ denotes a unit mass at the origin. The steeper the policy function is for a given attained service x , the more jobs on the average will be

found with that much service attained. This result is corroborated by the fact that the service rate of a job with x seconds of attained service is inversely proportional to $F'(x)$ [10].

A variety of performance criteria may be obtained from the average density of resident jobs. The integral of $n(x)$ over x from zero to infinity yields the average number N of jobs in the system. Clearly, the minimization of N and the minimization of the overall average residence time R result in identical policy functions since $N = \lambda R$ [7]. Discriminatory policies with respect to the treatment of short jobs versus long ones are obtained by minimizing (maximizing) the integral of the average density over a specified range of attained service. Of course, the conservation law in equation (20) still applies.

6. SPECIFIC POLICY FUNCTIONS

In this section we discuss the system behavior resulting from linear, exponential and composite policy functions. While the response functions are presented for arbitrary service time distributions, exponentially distributed service times have been assumed for the diagrams. Note that the mean residual life of an exponentially distributed service time is equal to its mean. Consequently, the term M in equation (19) is zero independent of the shape of the policy function and the overall mean residence time is equal to FIFO's.

The response function resulting from a linear policy function $F(x)=Cx$, where C is a constant, is given by

$$R(x) = U - \lambda ES^2 C/2 + x(1+\rho C). \quad (23)$$

Note that it is linear in x and that it depends only on the first two moments and not on the shape of the service time distribution. Due to our constraint $F(x) \leq x$, the validity of equation (23) is limited to $C \leq 1$. Thus, FIFO which is defined by $C=0$ is included. The response functions for $C=0, .4, 1$ are shown in Figure 2 for an M/M/1 system with a load of 75%. It is easily shown that the value of $R(ES^2/2ES)$ is independent of C ($ES^2/2ES$ is the mean residual life of the service time S). Although policy functions with $C > 1$ are not covered by our result, it is known that the feedback scheme is defined by $C \rightarrow \infty$ [10]. Its response function [5, p.174],

$$R(x) = x/(1-\lambda ES_{<x}) + \lambda ES_{<x}^2 / 2(1-\lambda ES_{<x})^2,$$

is also depicted in Figure 2. Varying C from 0 to ∞ , we note that a breakpoint occurs at $C=1$ where the response function loses its linearity and its invariant value at $x=ES^2/2ES$. The intuitive explanation, namely that the breakpoint separates the non-preemptive from the preemptive algorithms, has been provided by our theorem.

It is instructive to compare the linear policy function algorithms (linear policies) with the selfish scheduling algorithms [5, p.188]. In these algorithms, all jobs in the

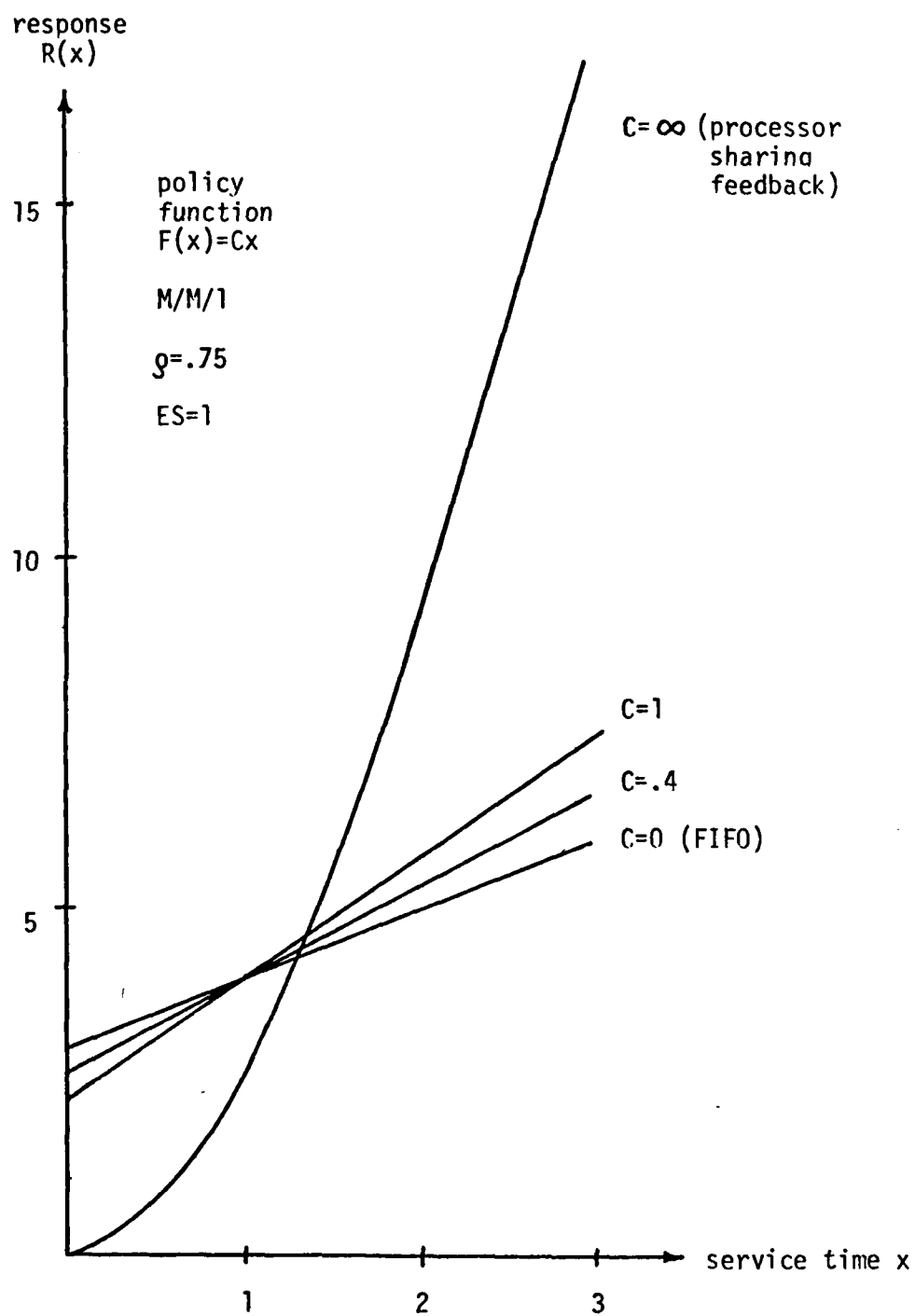


Figure 2. Response functions of linear policies for M/M/1, $\rho = .75$, ES=1.

system are divided into two groups: those in a "queue box" waiting for service, and those in a "service box" sharing the facility according to a given algorithm which is referred to as the raw scheduling algorithm. An arrival enters the queue box where its age (a numerical value) increases from zero at rate α . Similarly, the age of a job in the service box increases at rate β , where $\alpha \geq \beta \geq 0$. A job passes from the queue box to the service box when its age equals that of the jobs in the service box. When round robin and feedback algorithms are used as the raw scheduling algorithms, the selfish round robin (SRR) and the selfish feedback (SFB) algorithms result. Linear policies and SFB cover a (different) continuum of algorithms ranging from FIFO to feedback by varying C , $0 \leq C < \infty$, and β/α , $1 \geq \beta/\alpha \geq 0$, respectively. But while SFB permits preemption of the highest priority group by a new arrival throughout the entire range except for $\beta/\alpha = 1$, linear policies are non-preemptive for $C \leq 1$. Interestingly, the response functions in this non-preemptive range are identical to those of SRR with $1 \geq \beta/\alpha \geq \beta/(1+\beta)$. Of course, SRR is non-preemptive and approaches the round robin algorithm with $\beta/\alpha \rightarrow 0$. The linear policies thus form a hybrid between SFB and SRR. Furthermore, a linear policy with parameter C may itself be used as the raw scheduling algorithm for a selfish scheduling system with parameters α and β , thus defining a selfish linear policy. As it turns out, such a selfish linear policy with parameters α , β , $C \leq 1$ results in a

response function identical to that of a linear policy with parameter $C(1-\beta/\alpha)$. It can be shown that there is a good reason for this identity; the two algorithms are actually equivalent.

Next consider the class of exponential policy functions with $F(x)=C[1-\exp(-kx)]$. From the constraint $F(x)\leq x$ we have $Ck\leq 1$ and equation (17) becomes

$$R(x) = U + \lambda C \int_0^{\infty} \exp(-ky) G^C(y) dy + x - \rho C \exp(-kx).$$

The forms of the response functions for $k=1/3$ and $C=0,2,3$ are given in Figure 3 for a 75% loaded M/M/1 system. The density can be obtained from equation (22):

$$n(x) = \lambda [1 + \rho k C \exp(-kx)] G^C(x) + \lambda R(0) \delta(0).$$

The expression in brackets demonstrates that these exponential policies tend to keep waiting those jobs with little attained service. Holding kC constant, this tendency can be weakened by choosing a smaller value of k .

The composite policy functions of the form $F(x)=0$ for $x < m$ and $F(x)=x$ for $x \geq m$ are representative of the set of optimal policy functions which are obtained by minimizing the average overall residence time. The corresponding response functions,

$$R(x) = U - \lambda \int_m^{\infty} y G^C(y) dy + \rho F(x) + x,$$

are depicted in Figure 4 for $m=0,1,2$ and an M/M/1 system

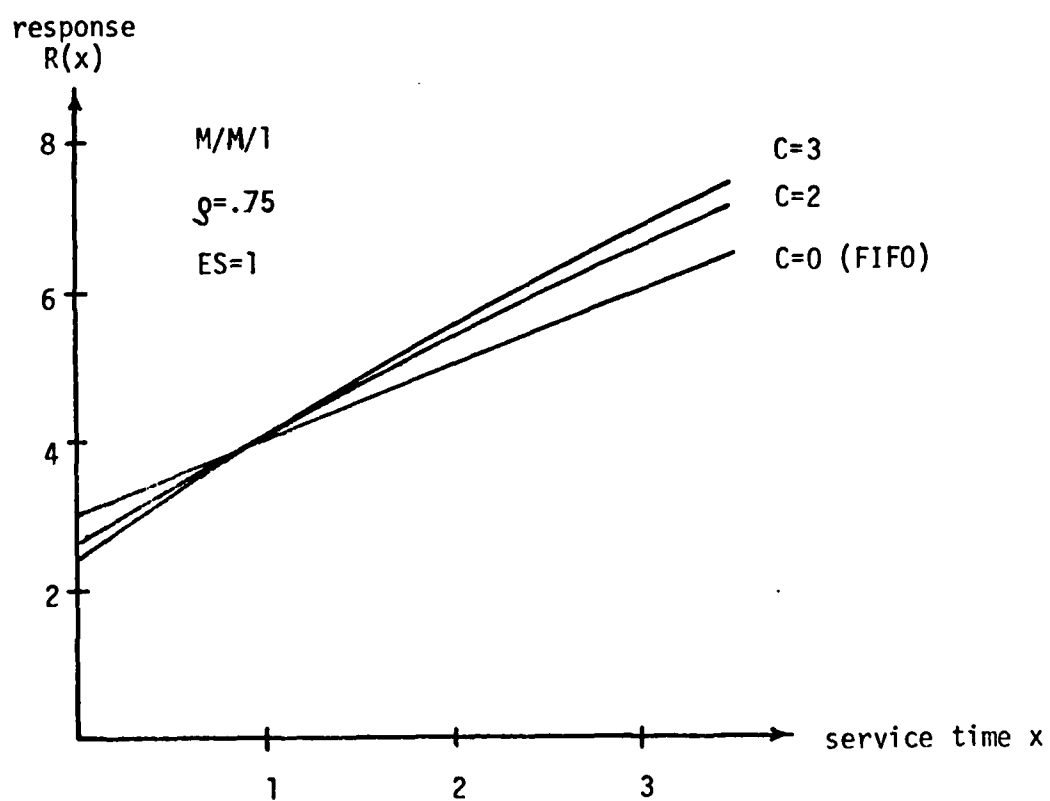
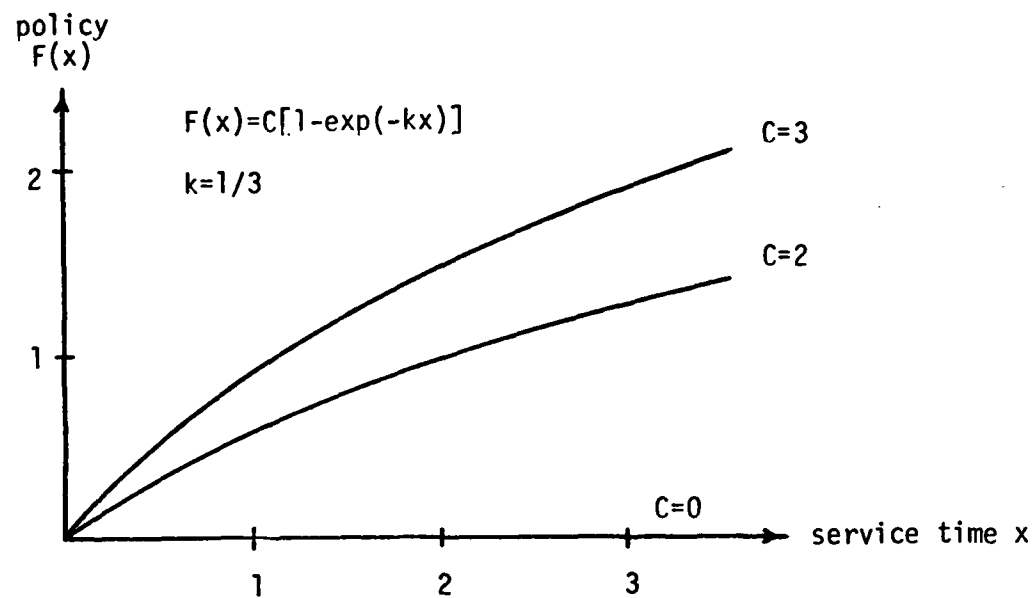


Figure 3. Exponential policy functions and their response for M/M/1, $\rho = .75$, $ES = 1$.

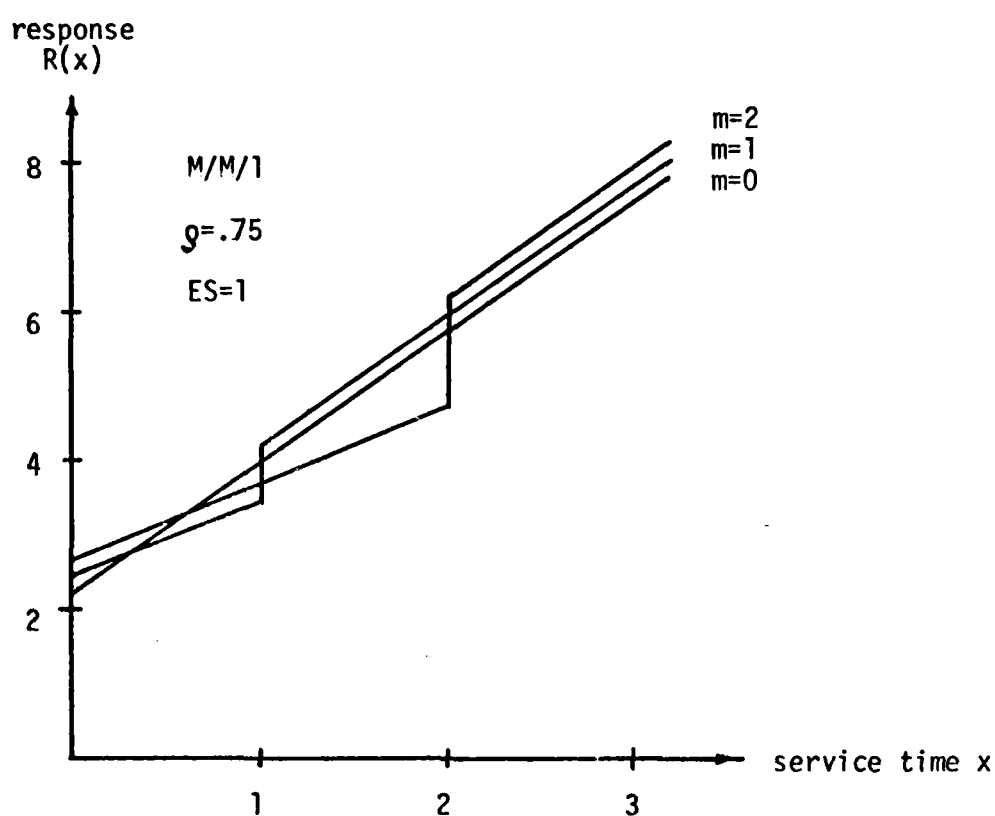
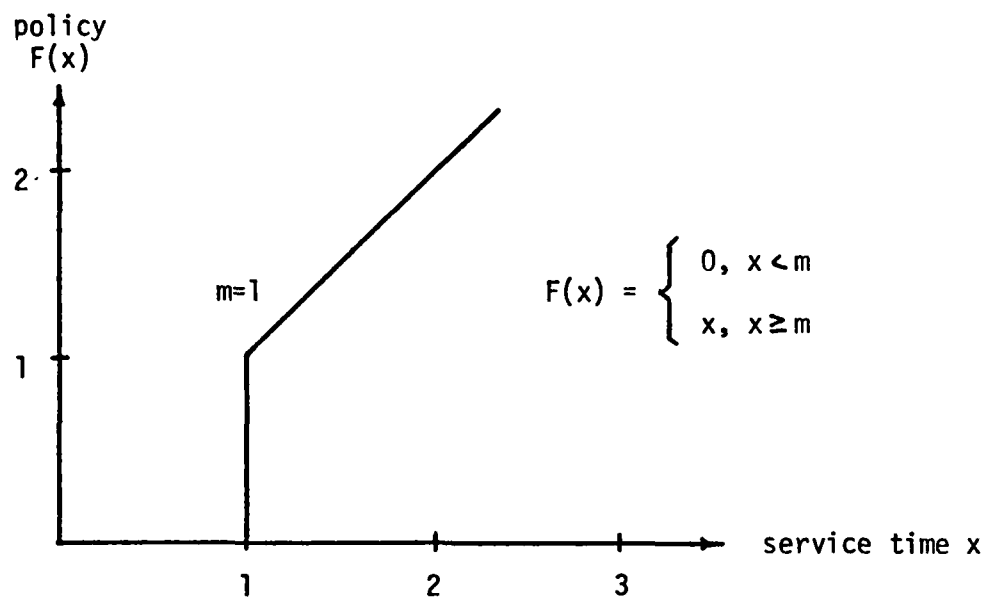


Figure 4. Composite policy functions and their response for $M/M/1$, $g=.75$, $ES=1$.

with a load of 75%. The jump of the response function at the value m of the service time is reminiscent of the jump resulting from the 2-level feedback algorithm [9]. This similarity does not come as a surprise when one considers that the defining policy function for 2-level feedback algorithms is given by $F(x)=0$ for $x < m$ and $F(x)=z$ for $x \geq m$, where $z \rightarrow \infty$ [10]. This (infinite) step function is itself closely related to a composite policy function. However, due to its height, it is not covered by our analysis which is constrained to functions of the form $F(x) \leq x$. The difference between the two algorithms can be explained in terms of the derivatives of their policy functions. A detailed discussion of policy function derivatives and their relation to processing rates is contained in [10].

Due to the parameters, each of the linear, exponential, and composite policies represents a (different) continuum of scheduling algorithms. In each case, the continuum includes $F(x)=0$ (FIFO) and $F(x)=x$. Linear policies, $F(x)=Cx$, cover this range with C varying from 0 to 1. Exponential policies, $F(x)=C[1-\exp(-kx)]$, have two parameters, C and k . $C=0$ results in FIFO. For $Ck=1$, the limit of $k \rightarrow 0$ approaches $F(x)=x$. Composite policies, $F(x)=0$ for $x < m$ and $F(x)=x$ for $x \geq m$, cover FIFO through $F(x)=x$ when m is varied from infinity to zero. The choices of a particular policy and its parameter(s) should be guided by some optimization criterion and will strongly depend on the given service time distribution. Current practice often ignores the dependence

of response on the service time distribution, possibly because of the lack of meaningful measurements. The FIFO end of a continuum is of interest for ES-MRLA distributed service times when the overall residence time is to be minimized. A policy close to $F(x)=x$ will treat short jobs more favorably. Composite policies satisfy both of these criteria when the mean residual service time is not bounded above.

The validity of the response functions in equation (17) is limited to policy functions constrained by $F(x) \leq x$, but the definitions of linear and exponential policies are not. (Neither is the definition of composite policy functions if multiplied by a constant C .) This constraint limits the favorable treatment of short jobs by assuring that the priority of a job cannot be higher than that of an earlier arrival. Without this constraint, the degree of discrimination is governed by the conservation law in equation (20) and the bounds in equation (21). In fact, linear policies result in the most discriminatory scheduling algorithm in favor of short jobs, the feedback algorithm, when the slope C of the policy function approaches infinity.

CONCLUSION

In this paper, the system behavior resulting from a broad class of policy functions is analyzed. The analysis is based on a work-conserving, processor-sharing M/G/1 queuing system and the system behavior is described in terms of the response function. To provide the necessary background, some basic properties of policy functions, including equivalence and normalization, have been briefly reviewed. Our analysis addresses the class of algorithms which are defined by $F(a) \leq a$, where the parameter a of the normalized policy function F is the attained service time. (Policy functions which have not been normalized belong to the same class if they satisfy $F(a) \leq F(0) + a$.) The main result, equation (17), gives the response functions for this class of algorithms. As expected, these response functions satisfy the conservation law and the tight bounds which have been established for time-sharing systems [5]. Linear, exponential, and composite policy functions are presented as examples for our general result. Criteria for the selection of a particular policy function are also discussed. If the overall mean response time is used as a performance measure, it is shown how the optimal policy function can be obtained as a function of the service time distribution. Of course, other performance measures, like the average number of resident jobs with less or more than a certain amount of attained service, may be chosen, but the closed form expressions of our results permit the solution of a variety

of different optimization problems.

The derivation of the response function involves the non-Markovian early work process. Its analysis was simplified by a decomposition method which is adaptable to other models. Due to this method, the derivation does not depend on the use of transforms and, thus, amplifies the details of the internal system behavior. The constraint $F(a) \leq a$ on the policy functions arises from a comparison of the busy periods of the unfinished work process and the early work process. Our theorem illustrates how the notion of preemption can be related to the busy periods of these two processes.

The mathematical simplicity of the main result is especially appealing. Nevertheless, policy functions which are not constrained by $F(a) \leq a$ should be analyzed since they behave differently with respect to preemption. Also, the consideration of memory requirements in the definition of a policy function would extend the applicability of the model to working set management problems. In any event, the relationship between policy functions $F(a) \leq a$, service time distributions, and system response has been established and can readily be used to optimize the response with respect to a given performance criterion.

GLOSSARY OF NOTATION

- F policy function, a (weakly) monotonic increasing function of attained service, $F(0)=0$, F may be discontinuous.
- F' its derivative, a non-negative function which may include Dirac's delta pulses.
- F^{-1} the inverse of F . For constant portions of F , i.e. $F(x)=c$ for $y \leq x \leq z$, the inverse is defined as $F^{-1}(c)=z$.
- $EX, E[X]$ the expectation of a random variable X .
- I interarrival time, an exponentially distributed random variable having mean $1/\lambda$.
- λ job arrival rate, $\lambda = 1/EI$.
- j job number, $j \geq 0$. Jobs are numbered in order of arrival.
- I_j time between arrivals of jobs $j-1$ and j ($j \geq 1$), I_j is exponentially distributed with mean $1/\lambda$.
- T_j arrival epoch of job j , $T_j = T_{j-1} + I_j$, $T_0 = 0$, $T_j = \sum_{k=1}^j I_k$.
- S service time, a random variable with distribution G .
- S_j service time requirement of job j . S_j has distribution G .
- G cumulative distribution function of S, S_j .
- g density function of S, S_j ; $g(s) = dG(s)/ds$.
 g may include Dirac's delta pulses.
- G^c complement of G , $G^c(s) = 1 - G(s)$, $dG^c(s)/ds = -g(s)$.
- $S_{\leq x}$ truncated service time, $S_{\leq x} = \min[S, x]$.
- n job number of the test job.

- x service requirement of the test job, $x=S_n$.
 t arrival epoch of the test job, $t=T_n$.
 L residence time of the test job, $L=R(t,x)$.
 a_j attained service of a resident job $j < n$ at the test job's departure epoch.
 i time between arrivals of the test job n and a late arrival (a job $j > n$), $i=T_j - t$.
 A attained service of a resident job $j > n$ at the test job's departure epoch.
 ρ system load, $\rho = \lambda ES$.
 $U(t)$ unfinished work in system at time t , a stochastic process.
 U time average of the unfinished work in system,

$$U = \lambda ES^2 / 2(1 - \rho).$$
 $R(t,x)$ residence time of the test job, $R(t,x) = W(t,x) + x$.
 $W(t,x)$ waiting time of the test job, $W(t,x) = R(t,x) - x$,

$$W(t,x) = V_E(t,x) + V_L(t,x).$$
 $V_E(t,x)$ early work encountered by test job.
 $V_L(t,x)$ late work encountered by test job.
 $V_\Delta(t,x)$ difference between unfinished work and early work,

$$V_L(t,x) = U(t) - V_E(t,x).$$
 $Q(x)$ time average of a process $Q(t,x)$ conditioned on x .
 Q overall time average of process $Q(t,x)$.
 $n(x)$ average density of jobs with x seconds of attained service.
 N time average of number of jobs in the system.

The following identity for the m -th moments of the truncated service time can be derived by an integration by parts:

$$ES_{<x}^m = \int_0^x s^m g(s) ds + x^m G^c(x) = \int_0^x ms^{m-1} G^c(s) ds.$$

By definition, $ES_{<x}^1 = ES_{<x}$ and $ES_{<x}^m$ approaches ES^m for $x \rightarrow \infty$.

REFERENCES

1. Athans, M., and Falb, P. L., Optimal Control, McGraw-Hill Book Co., New York, 1967.
2. Barlow, R. E., and Proschan, F., Mathematical Theory of Reliability, John Wiley and Sons, New York, 1965.
3. Bernstein, A. J., and Sharp, J. C., "A Policy-Driven Scheduler for a Time-Sharing System," Comm. ACM, 14, 2, 74-78 (1971).
4. Feller, W., An Introduction to Probability Theory and Its Applications, Volume II, Second Edition, John Wiley and Sons, New York, 1971.
5. Kleinrock, L., Queueing Systems, Volume II: Computer Applications, John Wiley and Sons, New York 1976.
6. Levin, R. et al., "Policy/Mechanism Separation in HYDRA," Proc. Fifth Symposium on Operating Systems Principles, University of Texas, Austin, 132-140 (1975).
7. Little, J. D. C., "A Proof for the Queuing Formula: $L = \lambda W$," Opns. Res., 9, 3, 383-387 (1961).
8. Lynch, H. W., and Page, J. B., "The OS/VS2 Release 2 System . Sources Manager," IBM Systems J., 13, 4, 274-291

(1974).

9. Ruschitzka, M., "System Resource Management in a Time Sharing Environment," Ph.D. dissertation, University of California, Berkeley, November 1973.

10. Ruschitzka, M., and Fabry, R. S., "A Unifying Approach to Scheduling," to appear in Comm. ACM in July 1977. Also available as Report SOSAP-TR-14, Dept. of Computer Science, Rutgers U., New Brunswick, New Jersey, June 1975.

11. Wolff, R. W., "Work-conserving Priorities," J. Appl. Probability, 7, 2, 327-337 (1970).

12. Wolff, R. W., "Time Sharing with Priorities," SIAM J. Appl. Math., 19, 3, 566-574 (1970).

REALIZATIONS OF SEQUENTIAL MACHINES
USING RANDOM ACCESS MEMORY

E.J. Wilkens

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research
Projects Agency of the Department of Defense under Grant #DAHC15-73-G6
to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the
author and should not be interpreted as necessarily representing the
official policies, either expressed or implied, of the Advanced
Research Projects Agency or the U. S. Government.

**REALIZATIONS OF SEQUENTIAL MACHINES
USING RANDOM ACCESS MEMORY****Abstract**

Modern large scale integration techniques, microcomputers, and programming techniques have made the classical realizations of sequential machines obsolete. These applications of sequential machines all occur in environments where random access memory, whether read-only or writable, is an extremely cost effective device. This paper presents a realization of a sequential machine which preserves the multiport branching characteristic of a sequential machine. A structure theory and a design technique are presented which allow an optimal memory size realization to be found.

I. INTRODUCTION

Among the many facets of system design, one which has been and will continue to be a fruitful area of study is the implementation of control. Control may be considered on various levels. Several examples are microprogrammed control of a central processing unit, microcomputer control of hardware systems, and logical control of programmed systems. This paper examines efficient realizations of sequential machines implemented as tables stored in random access memory, which makes application to any or all of the above control examples possible.

Sholl [1] has studied the application of a sequential machine to control of a microprogram, that part which determines the next address of the microprogram, as opposed to optimization of the control information field, which has been studied by Grasselli and Montanari [2]. (Unfortunately the word control is used at two levels here since in fact we are discussing the internal control of a microprocessor which in turn is being used to control the larger CPU.) Sholl developed a representation of a sequential machine used in place of the conventional next address field and branching operations in microprocessors to achieve a speed advantage. The representation to be presented here fits into Sholl's model, retaining the speed advantage but having different implementation and memory space characteristics.

A rapid rise in the use of microcomputer control of hardware systems is a result of an ever decreasing crossover point in the costs of microcomputers compared to TTL controllers. This crossover occurs in the range of 30 to 50 TTL circuits in today's technology [3]. As a result, the predominant users of

microcomputers are electronic design engineers [4]. Therefore, the use of sequential machines as a programming technique for microcomputers fits in quite well with the backgrounds of the typical users. Provision of a good specification language and an automatic optimizer provides a "higher level language" with inefficiency problems removed, since optimization of a sequential machine is easier than compiler code optimization.

Many authors have applied sequential machines to the logical control of large programs. For example, Millenson [5] describes their application to the control of psychological experiments. The realizations considered here are applicable to such programmed control.

The problem of assignment of binary variable codes to the internal states of a machine to allow its realization to approach some measure of optimality has received considerable attention, especially for implementations using combinational circuits and Flip-Flops [6]-[9]. Sholl [1] introduced the use of memory in the implementation of a sequential machine.

This paper continues the investigation of the use of memory in the implementation of a sequential machine, but with a representation with very different properties than Sholl's. The task is to code a reduced sequential machine into a format capable of being loaded into a random access memory. A table representation will be used since tables may be stored in random access memories efficiently.

Figure 1 shows a conceptual interpreter and table implementation of a sequential machine. An input causes the present state to be read from state storage. The input and present state form an index into the table containing the next-state and output functions. The next-state is read from the table and written into state storage, while the output is used appropriately.

In Sholl's realization, the interpreter is an address computation combinational network, with state and input as network inputs, and the address of the next-state, output pair as the network output. The state assignment has an effect on memory redundancy and network complexity. His approach sacrifices a small amount of redundancy in order to achieve a low level of network complexity. This is done by using a "minimum address variable assignment", which has one table entry for each defined state table cell. He has achieved a speed advantage over programmed or microprogram branch implementations since this technique may be considered a "multiport branch"[1].

The method considered in this paper decomposes the sequential machine into a sub-table per input, with an additional table of pointers to these sub-tables. Although the sequential machine is reduced, there may be pairs of states with identical entries in a single input sub-table. In addition, there may be some don't-care entries. It is assumed that the next-state and output entries are either both don't-care or both specified. Such a don't-care occurs when the input is externally constrained not to occur while the sequential machine is in a particular state. The output is never specified without the next-state since, if the transition can never occur, neither can the output, and conversely, if the transition occurs, an output is always significant. The output can be null, or no operation, which is distinct from don't-care.

Output or next-state only don't-cares are widely treated in the switching theory literature. However, in the state table applications mentioned above and others, such don't-cares were not used. Since these single don't-cares give added complexity to the problem, they are not treated here.

If the input is known, then there is redundancy in the information provided by the state. For example, it is never necessary to identify a state whose entry is don't-care, nor is it necessary to distinguish between two states with the same entries. Providing memory locations for this redundant information is wasteful. A state encoding scheme that allows identification of a unique entry in the table is desirable. That is, the original sequential machine is transformed to a table with only one occurrence of each distinct entry in each input sub-table.

Consider Figure 2. The input sub-table corresponding to input 1 must contain four entries, one for each of 1A, 3A, 5A and 5B. Similarly, the input 4 sub-table must contain two entries. The state number no longer provides a direct index to the sub-tables. For example, if the machine is in state 7 and input 1 is received, there is no 7th entry in the input 1 sub-table. A mapping algorithm from the state to the input sub-table indices is needed. The method adopted here for its simplicity of use is to encode the states using binary variables $y_1 y_2 \dots y_n$ such that for each input there exists a substring of these binary variables which can be used as an index into the corresponding input sub-table. Figure 3 shows an appropriate encoding of the states of Figure 2. Since the I_1 sub-table has four entries, the I_1 substring should have four distinct codes, which are 00, 01, 10, and 11 under variables $y_2 y_3$. Note that the I_2 substring $y_1 y_2$ has exactly three values. In addition, values of $y_1 y_2$ for any two states are the same if and only if the entries of those two states are the same or one or both is don't-care under input 2 in Figure 2. Thus states 1 and 4 have the $y_1 y_2$ value 00. Under input 2 state 1 is don't care and state 4 is 5A. Figure 4 shows the tables which ultimately realize the machine of Figure 2.

As an example, consider the machine to be in state 5. Thus 1010 is in the state storage. If input I_2 is received, the y_1y_2 variables are read and used to index the I_2 sub-table. The binary 10 is equivalent to decimal 2, thus 2 is added to the sub-table pointer for input 2, and TABLE +6 is read. The entry 0001 A is the encoding for 4A, which is the proper state 5, input 2 entry. If input I_4 is received, bit y_1 is 1, 1 to be added to the I_4 sub-table pointer, TABLE +11. Thus, TABLE +12 is read, giving 1010 A, which is equivalent to 5A. In each case, the state portion of the entry is written in its entirety back into state storage. The output might be the address of an output routine or an index into a branch table, etc. The assumption made here is that the interpreter for the substring encoding is either software or a circuit which for each input masks the appropriate bits, shifts them to the right, and uses them as an index into the sub-table. By using such a circuit or software, there is less custom designed circuitry, none in the case of the software interpreter, and a circuit amenable to possible LSI integration with only the final mask and shift quantities to be determined for the individual state table, perhaps to be loaded into a Read Only Memory. Figure 5 shows a possible hardware implementation, while Figure 6 shows a PL/I implementation. Note that this implementation retains the same "multiport branch" capability as the Sholl realization.

A technique used in the work presented here, which is general enough to appear applicable to other problems, is the method of treating don't-care conditions. The approach followed is to formulate representations of information by means of constructs called partitions-with-don't-cares (PDCs) which include all specified information, but allow nonspecified information (don't cares) to take on all possible values. These constructs are constrained to be compact (nonenumerative) and to possess a set of rules to allow enumeration of the set of fully specified entities represented by the

construct. Rules are also provided for generation of such constructs by operations on other such constructs. Such a technique, if utilized fully, would prevent enumeration as a synthesis technique. In the work presented this technique is extensively used, but it has not been applied to all the synthesis steps.

The PDC construct is used throughout the synthesis procedure. In places, operators produce only a single PDC as a result. In other places, use must be made of limited enumeration. These techniques make possible the optimization of larger state tables by a programmed version of the procedure than has been previously reported. A program, which does not yet incorporate all the features of the procedure presented here has successfully optimized a 9 state, 5 input machine in 2.3 seconds, and the 7 state, 4 input machine shown in this paper in 1.5 seconds in an IBM 370/168.

Introduction of all the features presented here should speed it up considerably. This program does, however, contain some enumeration pruning techniques which are not presented here in order to allow a sufficiently small example whose solution may be completely described to serve as an expository vehicle. The techniques not presented would not prune the present example to any significant degree. They will be presented in a subsequent paper dealing with such techniques for computer implementation of the synthesis procedure.

II. PARTITIONS WITH DON'T-CARES

This section introduces concepts that will be used in finding the state encoding introduced in Section I. These concepts are extensions of partition theory [6], and with which reader familiarity is assumed.

Definition 1. A partition with don't-cares (PDC) of a set S is a collection of disjoint subsets of S whose set union is S , one subset of which is distinguished and may be empty.

Conventions used in denoting PDCs are that blocks are enclosed in parentheses and the distinguished block is enclosed in $\langle \rangle$. For example $(1,2)(3,5)(4,6)\langle 7,8 \rangle$ is a 3-block PDC on $S = \{1,2,3,4,5,6,7,8\}$ with a distinguished block $\langle 7,8 \rangle$ and blocks $(1,2)$, $(3,5)$, $(4,6)$. Blocks of a PDC π are also denoted by $B_i(\pi)$. $B_d(\pi)$ denotes the distinguished block. A PDC with an empty distinguished block is a partition. $\#(\pi)$ is the number of non-distinguished blocks in π . The motivation for this definition of a PDC is the fact that many problems involving don't-care conditions induce equivalence relations on sets, with the exception of those members of the sets designated by don't-cares. Since a true partition would specify information about the members to be considered don't-care, it is desirable to define a structure with the ability to leave some elements unspecified. Furthermore, it is considered that the equivalence relation on the specified members of the set determines the maximum of information the PDC should generate. That is, the distinguished block is to be treated not as a true block, but as a subset which may be distributed arbitrarily over the specified blocks. Thus the PDC π in reality defines a set of partitions, denoted by $G(\pi)$, generated by distributing the distinguished block over the

other blocks of the PDC in all possible ways.

Definition 2. A PDC π_1 is less specified than a PDC π_2 ($\pi_1 \leq_s \pi_2$) if and only if

- 1) Each nondistinguished block of π_1 is contained in a nondistinguished block of π_2 and
- 2) Every nondistinguished block of π_2 contains exactly one non-distinguished block of π_1 .

Definition 3. The set of partitions generated by π , $G(\pi)$, is $\{\tau | \tau \text{ is a partition and } \pi \leq_s \tau\}$.

Thus, if $\pi_1 = (1,4)(2)<3,5>$ then $G(\pi_1)$ is the set of partitions $(1,3,4,5)(2)$, $(1,4)(2,3,5)$, $(1,3,4)(2,5)$, and $(1,4,5)(2,3)$.

In general, problems with don't-care conditions generate PDCs initially. Good encodings for the problem under consideration can then be achieved by assigning don't cares judiciously, without interfering with the original information content. Thus the specification ordering gives a method of assigning don't-cares without destroying original information. If $\pi_1 \leq_s \pi_2$ then $G(\pi_2) \subseteq G(\pi_1)$. Thus, specifying additional elements reduces the size of the set generated.

Definition 4. The relation less than or equal to holds between two PDCs π_1 and π_2 ($\pi_1 \leq \pi_2$) if and only if 1) for every $B_i(\pi_1)$, $i \neq d$, there exists a $B_j(\pi_2)$ such that $B_i(\pi_1) \subseteq B_j(\pi_2) \cup B_d(\pi_2)$, and 2) if there exists a set $\{B_j(\pi_2) | j \neq d \text{ and } B_j(\pi_2) \subseteq B_d(\pi_1)\}$ then there exists a set $\{B_i(\pi_1) | i \neq d \text{ and } B_i(\pi_1) \subseteq B_d(\pi_2)\}$ of equal or greater cardinality than $\{B_j(\pi_2)\}$.

If the π_1 and π_2 are partitions, this definition reduces to the conventional ordering relation between two partitions.

The following theorems show that $\pi_1 \leq \pi_2$ if and only if in the sets $G(\pi_1)$ and $G(\pi_2)$ of fully specified partitions there exists at least one pair (τ_1, τ_2) , $\tau_1 \in G(\pi_1)$ and $\tau_2 \in G(\pi_2)$, such that $\tau_1 \leq \tau_2$.

Theorem 1 If τ_1 and τ_2 are partitions and π_1 and π_2 are PDCs such that $\tau_1 \leq \tau_2$, $\tau_1 \in G(\pi_1)$, and $\tau_2 \in G(\pi_2)$, then $\pi_1 \leq \pi_2$.

Proof: Condition 1:

$\tau_1 \leq \tau_2$ implies that for every $B_i(\tau_1)$ there exists a $B_j(\tau_2)$ such that $B_i(\tau_1) \subseteq B_j(\tau_2)$. Therefore $B_i(\pi_1) \subseteq B_i(\tau_1) \subseteq B_j(\tau_2) \subseteq B_j(\pi_2) \cup B_d(\pi_2)$.

Condition 2: Consider $B_j(\pi_2) \subseteq B_d(\pi_1)$. $\tau_1 \leq \tau_2$ implies that for every $B_j(\tau_2)$ there exists a $B_i(\tau_1)$ such that $B_i(\tau_1) \subseteq B_j(\tau_2)$. Then $B_i(\pi_1) \subseteq B_i(\tau_1) \subseteq B_j(\tau_2) \subseteq B_j(\pi_2) \cup B_d(\pi_2) \subseteq B_d(\pi_1) \cup B_d(\pi_2)$. Therefore $B_i(\pi_1) \subseteq B_d(\pi_2)$. Since $B_i(\pi_1)$ and $B_j(\pi_2)$ are both contained in $B_j(\tau_2)$ and no other block of π_2 is contained in $B_j(\tau_2)$, there are at least as many $B_i(\pi_1)$ contained in $B_d(\pi_2)$ as there are $B_j(\pi_2)$ contained in $B_d(\pi_1)$.

Theorem 2 If π_1 and π_2 are PDCs such that $\pi_1 \leq \pi_2$, then there exist partitions τ_1 and τ_2 such that $\tau_1 \in G(\pi_1)$, $\tau_2 \in G(\pi_2)$, and $\tau_1 \leq \tau_2$.

Proof: Consider $B_j(\pi_2)$ such that $B_j(\pi_2) \subseteq B_d(\pi_1)$. There is a non-empty set $A = \{B_i(\pi_1) \mid B_i(\pi_1) \cap B_j(\pi_2) \neq \emptyset\}$. Let $B_j(\tau_2) = \bigcup A$. Let $B_i(\tau_1) = B_i(\pi_1)$ for all $B_i(\pi_1) \in A$. Finally, add each $s \in B_j(\pi_2) \cap B_d(\pi_1)$ arbitrarily to one of the blocks $B_i(\tau_1)$ formed from A . $B_i(\tau_1) \subseteq B_j(\tau_2)$ for all blocks of τ_1 and τ_2 formed in this manner.

Consider $B_j(\pi_2)$ such that $B_j(\pi_2) \subseteq B_d(\pi_1)$. For each such $B_j(\pi_2)$ choose a different block $B_i(\pi_1)$ such that $B_i(\pi_1) \subseteq B_d(\pi_2)$. There are always more such $B_i(\pi_1)$ than $B_j(\pi_2)$ since $\pi_1 \leq \pi_2$. Let $B_i(\tau_1) = B_j(\tau_2) = B_i(\pi_1) \cup B_j(\pi_2)$. Obviously $B_i(\tau_1) \subseteq B_j(\tau_2)$ for all blocks thus formed.

Some blocks $B_i(\pi_1)$ may remain which have not yet been included in a block of τ_1 and such that $B_i(\pi_1) \subseteq B_d(\pi_2)$. Add each such $B_i(\pi_1)$ to any existing block of τ_2 and let $B_i(\tau_1) = B_i(\pi_1)$. Thus there exists a $B_j(\tau_2)$ such that $B_i(\tau_1) \subseteq B_j(\tau_2)$.

Finally, only members of $B_d(\pi_1) \cap B_d(\pi_2)$ remain to be assigned to a block of τ_1 and τ_2 . Add each $s \in B_d(\pi_1) \cap B_d(\pi_2)$ to any block $B_i(\tau_1)$ and to the block $B_j(\tau_2)$ which contains $B_i(\tau_1)$. Thus the resulting block of τ_1 is still contained in the block of τ_2 .

Each block of τ_1 and τ_2 contains, by the method of construction, only a single block of π_1 and π_2 respectively plus elements of $B_d(\pi_1)$ and $B_d(\pi_2)$ respectively. Thus, $\tau_1 \in G(\pi_1)$ and $\tau_2 \in G(\pi_2)$. Since for each block $B_i(\tau_1)$ there exists a $B_j(\tau_2)$ such that $B_i(\tau_1) \subseteq B_j(\tau_2)$, $\tau_1 \leq \tau_2$.

The next theorem shows that condition 2 of the definition of $\pi_1 \leq \pi_2$ is necessary for the existence of the generated partitions for which $\tau_1 \leq \tau_2$ holds.

Theorem 3 If π_1 and π_2 are PDCs such that for every $B_i(\pi_1)$, $i \neq d$, there exists a $B_j(\pi_2)$ such that $B_i(\pi_1) \subseteq B_j(\pi_2) \cup B_d(\pi_2)$, and if the set $\{B_j(\pi_2) \mid B_j(\pi_2) \subseteq B_d(\pi_1)\}$ is of greater cardinality than the set $\{B_i(\pi_1) \mid B_i(\pi_1) \subseteq B_d(\pi_2)\}$, then there is no pair (τ_1, τ_2) such that $\tau_1 \in G(\pi_1)$, $\tau_2 \in G(\pi_2)$, and $\tau_1 \leq \tau_2$.

Proof: Each of the blocks of the set $\{B_j(\pi_2)\}$ mentioned above must be combined with a block of the set $\{B_i(\pi_1)\}$ mentioned above in both τ_1 and τ_2 so that $B_i(\tau_1) \subseteq B_j(\tau_2)$. However, there are more such blocks of π_2 . Thus, at least one block of τ_2 has no block of π_1 contained in it. Since each block of $\tau_1 \in G(\pi_1)$ contains one block of π_1 , no block of τ_1 is contained in that block of τ_2 . Thus $\tau_1 \not\leq \tau_2$ for all $\tau_1 \in G(\pi_1)$ and $\tau_2 \in G(\pi_2)$.

Theorem 4 If $\pi_1 \leq_s \pi_2$ then $\pi_2 \leq \pi_1$.

Proof: $\pi_1 \leq_s \pi_2$ implies that for every $B_i(\pi_1)$ there exists a unique $B_j(\pi_2)$ such that $B_i(\pi_1) \subseteq B_j(\pi_2)$. Since $B_j(\pi_2)$ is unique, $B_j(\pi_2) \subseteq B_i(\pi_1) \cup B_d(\pi_1)$. The sets $\{B_i(\pi_1) \mid B_i(\pi_1) \subseteq B_d(\pi_2)\}$ and $\{B_j(\pi_2) \mid B_j(\pi_2) \subseteq B_d(\pi_1)\}$ are empty. Therefore $\pi_2 \leq \pi_1$.

III. FORMALIZATION OF TABLE REALIZATION

In this section the concepts of a table realization are formalized. A "don't-care" will be denoted by "-" and a null output by ϕ .

Definition 5 A sequential machine is a 5-tuple $\langle S, I, O, \delta, \lambda \rangle$, where S is a finite set of states, I is a finite set of inputs, O is a finite set of outputs, δ is a function from $S \times I$ into $S \cup \{-\}$, and λ is a function from $S \times I$ into $O \cup \{-\}$, with the restriction that, for all $s \in S$ and $i \in I$, $\delta(s, i) = -$ if and only if $\lambda(s, i) = -$.

The restriction on δ and λ reflects the informal discussion of the previous section which stated that transitions and outputs should be specified or unspecified together and not independently.

Definition 6. A table realization of a sequential machine $\langle S, I, O, \delta, \lambda \rangle$ is a triple $\langle J, \xi, T \rangle$, where J is a set of index sets, J_i , each of which is a finite subset of integers, for each $i \in I$, ξ is a set of state mapping functions ξ_i for each $i \in I$ from S onto J_i , and T is a set of input sub-table functions T_i for each $i \in I$ from J_i into $S \times O$, subject to the constraints that 1) for any i , $\xi_i(s_j) = \xi_i(s_k)$ implies that either $\delta(s_j, i) = \delta(s_k, i)$ and $\lambda(s_j, i) = \lambda(s_k, i)$ or one of the δ 's is a don't-care, and 2) for all $i \in I$ and all $j \in J_i$, $T_i(j) = (\delta(s, i), \lambda(s, i))$ where s is such that $\xi_i(s) = j$ and $\delta(s, i)$ is not a don't-care.

The set T is the set of input subtables, each entry of which is a next-state and an output. Thus the function T_i from J_i into $S \times O$ is simply the natural function from a table index onto the table entry. The J_i are

the sets of indices used for each input subtable and ξ_i is a function from the state of the sequential machine onto the proper index to be used for that state under input i . Figures 3 and 4 illustrate a table realization of Figure 2. $J_1 = J_3 = \{0,1,2,3\}$, $J_2 = \{0,1,2\}$, $J_4 = \{0,1\}$ as can be seen from either the state encoding or the input subtables. The ξ_i functions are given by the state encoding by reading the proper substring of the state for i . Thus $\xi_1(1) = \xi_1(3) = \xi_1(4) = 0$, $\xi_1(5) = \xi_1(6) = 1$, $\xi_1(2) = 2$, $\xi_1(7) = 3$. The usual state table specification of a sequential machine is a table realization in which $J_i = S$ for all i , the ξ_i functions are identity functions, and each T_i is simply the i 'th column of the state table. It is clear from the example given that the T_i function is intended to be realized by a table, while the ξ_i is intended to be realized by an algorithm. The reading of a substring of the state variables is such an algorithm.

Definition 7. Each indexing function ξ_i induces an input table partition τ_i on the state set S of a sequential machine such that $s_j, s_k \in B_\ell(\tau_i)$ if and only if $\xi_i(s_j) = \xi_i(s_k)$.

Definition 8. For each input i of a sequential machine, δ and λ induce an input PDC π_i on the state set S such that

- 1) $s_j, s_k \in B_\ell(\pi_i)$ if and only if $\delta(s_j, i)$ and $\delta(s_k, i)$ are both specified and $\delta(s_j, i) = \delta(s_k, i)$ and $\lambda(s_j, i) = \lambda(s_k, i)$, and
- 2) $s_j \in B_d(\pi_i)$ if and only if $\delta(s_j, i) = -$.

Input table partitions are a means of analysis of table realizations of sequential machines by giving an algebraic characterization of the realization. On the other hand the input PDC introduce a synthesis requirement, as will be shown below. That is, the π_i 's are PDCs to which a realization's τ_i 's will have to conform. It is shown below that any table realization must have $\tau_i \leq \pi_i$, while for a minimal table entry table realization $\pi_i \leq_s \tau_i$.

Theorem 5 A partition τ_i is an input table partition induced by ξ_i of a sequential machine $\langle S, I, 0, \delta, \lambda \rangle$ if and only if $\tau_i \leq \pi_i$, where π_i is the input PDC generated by δ and λ under input i .

Proof: If τ_i is an input table partition then $s_j, s_k \in B_j(\tau_i)$ implies $\xi_i(s_j) = \xi_i(s_k)$. This in turn implies that either $\delta(s_j, i) = \delta(s_k, i)$ and $\lambda(s_j, i) = \lambda(s_k, i)$, in which case $s_j, s_k \in B_m(\pi_i)$ for some $m \neq d$ or $\delta(s_j, i)$ or $\delta(s_k, i)$ is don't care, in which case s_j or s_k is in $B_d(\pi_i)$. Therefore $B_j(\tau_i) \subseteq B_m(\pi_i) \cup B_d(\pi_i)$ and, since $B_d(\tau_i) = \emptyset$, $\tau_i \leq \pi_i$.

If $\tau_i \leq \pi_i$ then, for every j , there exists a k such that $B_j(\tau_i) \subseteq B_k(\pi_i) \cup B_d(\pi_i)$. If $s_l, s_m \in B_j(\tau_i)$ then either 1) both $s_l, s_m \in B_k(\pi_i)$ in which case $\delta(s_l, i) = \delta(s_m, i)$ and $\lambda(s_l, i) = \lambda(s_m, i)$; or 2) one or the other or both s_l or $s_m \in B_d(\pi_i)$, in which case $\delta(s_l, i) = -$ or $\delta(s_m, i) = -$. Therefore τ_i is generated by a ξ_i component of a table realization.

Theorem 6 The number of table entries in T to realize a sequential machine is minimum if and only if, for every i , $\pi_i \leq_s \tau_i$.

Proof: The number of table entries in T is the sum of the number of blocks of the τ_i partitions. Each τ_i and π_i are only dependent on δ and λ restricted to input i , and are therefore independent of other τ 's and π 's. Thus the number of blocks of each τ_i may be minimized independently.

If τ_i is an input table partition then $\tau_i \leq \pi_i$. Therefore τ_i has at least as many blocks as π_i . If $\pi_i \leq_s \tau_i$ then τ_i has exactly the same number of blocks as π_i , which is the minimum possible. Also, $\pi_i \leq_s \tau_i$ implies $\tau_i \leq \pi_i$, and τ_i results in a minimum entry table.

If $\tau_i \leq \pi_i$ and τ_i has a minimum number of blocks, then the number of blocks of τ_i and π_i are the same. Also, for each $B_j(\tau_i)$ there exists a $B_k(\pi_i)$ such that $B_j(\tau_i) \subseteq B_k(\pi_i) \cup B_d(\pi_i)$. Since the number of blocks are the same, the j - k correspondence is unique. Therefore for every $B_k(\pi_i)$ there exists a $B_j(\tau_i)$ such that $B_k(\pi_i) \subseteq B_j(\tau_i)$ and each $B_j(\tau_i)$ contains exactly one block of π_i . Thus $\pi_i \leq_s \tau_i$.

Theorems 5 and 6 show that in order to find a partition τ_i to give a minimum size table T , any member of $G(\pi_i)$ may be chosen for each input PDC π_i .

Definition 9. An encoding $E(\rho)$ of a partition ρ on S is a binary matrix of dimensions $\#(S) \times e$, such that s_i and s_j are members of $B_k(\rho)$ if and only if row i and row j of $E(\rho)$ are identical.

The encoding $E(S)$ is a particular case of $E(\rho)$ where ρ is the trivial zero partition of S (each $s \in S$ is in its own block).

Definition 10 An encoding of a string of partitions $E(\rho_1 \rho_2 \dots \rho_k)$, is formed from the encodings $E(\rho_i)$ of the individual partitions as follows:

- 1) $E(\rho_1 \rho_2 \dots \rho_k) = E(\rho_1)$ if $k = 1$
- 2) $E(\rho_1 \rho_2 \dots \rho_k) = E(\rho_1 \rho_2 \dots \rho_{k-1})$ augmented by $E(\rho_k)$.

By convention, a string $p_1 p_2 \dots p_k$ will be informally referred to as an encoding, while what is meant is $E(p_1 p_2 \dots p_k)$.

A minimum table entry realization may be easily constructed for any sequential machine. Arbitrarily choose a $\tau_i \in G(\pi_i)$ for each $i \in I$. Choose a minimum bit $E(\tau_i)$. The number of bits needed for each row of $E(\tau_i)$ is $\lceil \log_2(\#(\pi_i)) \rceil$. Let $E(S) = E(\tau_1 \tau_2 \dots \tau_{\#(I)})$. The substring contributed by each τ_i is used to index each input subtable T_i .

Figure 7 shows such an encoding applied to the machine of Figure 2. The machine which was encoded in four bits in Figure 3 has been encoded in seven bits in Figure 7 because no sharing of bits was used. Finally, Figure 8 lists the input table partitions for Figure 3.

Since each π_i of Figure 7 is less specified than the corresponding τ_i of Figure 8, the table realization of Figure 3 is a minimum entry table realization. The primary means of reducing the number of bits in the encoding, and consequently in each entry of the table, is to share the bits of the encoding among several inputs. It may also be noted that the τ_i 's of Figure 3 are very different from those of Figure 7. It is the judicious choice of the τ_i 's that allows the sharing of bits. The following sections describe methods of choosing τ_i 's to give maximum or near maximum sharing of bits.

By using the concepts developed thus far, several bounds on the resulting table realization may be easily calculated:

Table size

Size of $E(S)$ with
min. table size

Maximum

$\#(S) \times \#(I)$

$\sum_{i \in I} \lceil \log_2(\#(\pi_i)) \rceil$

Minimum

$\sum_{i \in I} \#(\pi_i)$

$\lceil \log_2(\#(S)) \rceil$

IV. SHARING OF CODE BITS BETWEEN INPUT PDCS

So far, sharing of code bits has been mentioned, but only informally. Sharing of code bits between input PDCs must be done at a finer level than input partitions, and so a suitable algebraic structure is needed. In order to analyze and synthesize the sharing of partitions in the encoding of two or more π 's, operations on partitions must be defined. It is assumed that conventional partition operations and properties are understood by the reader [6].

Definition 11. An encoding of a string of partitions $E(\rho_1\rho_2\dots\rho_k)$ is said to be a minimum table-size subset encoding of a sequential machine if and only if for each input PDC π_i there exists a subset R_i of $\{\rho_1,\dots,\rho_k\}$ called a row set such that $\pi_i \leq_s \prod_{\rho_j \in R_i} \rho_j$.

Definition 12. A subset encoding $\rho_1\rho_2\dots\rho_k$ of input PDCs π_1 through π_n , where n is the number of inputs, defines k sets of PDCs, called column sets ($C_j, 1 \leq j \leq k$) such that $C_j = \{\pi_i | \rho_j \in R_i\}$.

A row set R_i is simply the set of ρ s used to encode a specific π_i , and a column set C_j is simply the π s in whose encoding ρ_j is used.

In the example of Figure 3, ρ_1 through ρ_4 are each induced on the state set by each of the bits of the encoding y_1 through y_4 respectively.

$$\rho_1 = (1,2,4,6,7)(3,5)$$

$$\rho_2 = (1,3,4,5,6)(2,7)$$

$$\rho_3 = (1,2,3,4)(5,6,7)$$

$$\rho_4 = (1,2,3,5)(4,6,7)$$

$$\begin{array}{lll}
 R_1 = \{\rho_2, \rho_3\} & \pi_1 \leq_s \rho_2 \cdot \rho_3 & C_1 = \{\pi_2, \pi_4\} \\
 R_2 = \{\rho_1, \rho_2\} & \pi_2 \leq_s \rho_1 \cdot \rho_2 & C_2 = \{\pi_1, \pi_2\} \\
 R_3 = \{\rho_3, \rho_4\} & \pi_3 \leq_s \rho_3 \cdot \rho_4 & C_3 = \{\pi_1, \pi_3\} \\
 R_4 = \{\rho_1\} & \pi_4 \leq_s \rho_1 & C_4 = \{\pi_3\}
 \end{array}$$

Definition 13. A minimum bit, minimum table-size subset encoding of a sequential machine is a minimum table-size substring encoding such that

- 1) $\sum_i \lceil \log_2 \#(\rho_i) \rceil$ is less than or equal to the same summation for all other minimum table-size substring encodings of the same machine, and
- 2) for all π_i , $\lceil \log_2 \#(\pi_i) \rceil = \sum_{\rho_j \in R_i} \lceil \log_2 \#(\rho_j) \rceil$.

Lemma 1[6]. If $\rho = \rho_1 \cdot \rho_2$ then $\rho \leq \rho_1$, $\rho \leq \rho_2$.

Any subset of an encoding may be used for an I_i . For example, in $\rho_1 \rho_2 \rho_3 \rho_4$, if $\rho_1 \cdot \rho_2 = \tau_1$ then $\rho_1 \rho_2$ may be used for I_1 . A possible use of $\rho_1 \rho_2 \rho_3 \rho_4$ may be shown diagrammatically as:

	ρ_1	ρ_2	ρ_3	ρ_4
τ_1	0	1	1	0
τ_2	1	1	0	0
τ_3	0	0	1	1
τ_4	1	0	0	0

The concatenated partitions of the encoding are listed across the top, the input table partitions on the side, and the subset corresponding to each input table partition has a 1 under each ρ used for it. This diagram is called an encoding profile.

Lemma 2[6]. The sum of two partitions $\rho_1 + \rho_2$ is the least upper bound of ρ_1 and ρ_2 .

Theorem 7. If a partition ρ can be shared by input partitions τ_i and τ_j then $\tau_i + \tau_j \leq \rho$.

Proof: Since the sharing of ρ implies there exists τ_i' and τ_j' such that $\tau_i = \tau_i' \cdot \rho$ and $\tau_j = \tau_j' \cdot \rho$, it follows that $\tau_i \leq \rho$ and $\tau_j \leq \rho$. Since $\tau_i + \tau_j$ is the least upper bound of τ_i and τ_j , $\tau_i + \tau_j \leq \rho$.

Theorem 8. Consider a minimum table-size subset encoding $E(\rho_1 \rho_2 \dots \rho_k)$.

Let $\tau_i = \prod_{\rho_j \in R_i} \rho_j$ for every i in the set of inputs. For each component partition ρ_j of the encoding,

$$\sum_{\pi_i \in C_j} \tau_i \leq \rho_j$$

Proof: Since ρ_j appears as part of the encoding of all τ_i where $\pi_i \in C_j$, $\tau_i \leq \rho_j$ for all $\pi_i \in C_j$. Since the sum of partitions is the least upper bound of the partitions, and ρ_j is an upper bound on all the τ_i for $\pi_i \in C_j$, $\sum_{\pi_i \in C_j} \tau_i \leq \rho_j$.

The theorems of this section thus far have primarily dealt with partitions induced by an encoding, and the properties of these partitions. They give little help in finding such an encoding from the input PDCs. Operations that lead toward such an encoding are considered next.

In order to discover the ways in which two PDCs, π_1 and π_2 , may share a partition ρ in an encoding, enumeration of every ρ greater than or equal to

the sums $\tau_k + \tau_l$ for all (τ_k, τ_l) in $G(\pi_1) \times G(\pi_2)$ may be performed. However, a "closed form" of this set of partitions, just as a PDC π_i is a "closed form" of the set $G(\pi_i)$, is of greater value. It is this set of partitions that is considered the sum of two PDCs. This sum must preserve the characteristics which are considered essential in the encoding desired. Specifically, minimum table size is to be preserved. The major problem is that each of the two PDCs to be added have, in general, different distinguished blocks. Because of this fact, every block of $\pi_1 + \pi_2$ must contain at least one block of π_1 and one block of π_2 . This requirement arises from the objective of minimum table size. If, by taking the sum of two PDCs, a block is introduced in which no block of one PDC is contained, then no matter what partition the sum is combined with by the product operation, that block will stay distinct, since the product operation tends to split blocks, not join them. Therefore, in the final encoding there will be a block which is entirely don't-care for that input, and a table entry will be wasted. Stated more formally, $\pi \not\subseteq_S \tau$ since one block of τ has no block of π contained in it.

The definition of the sum of two PDCs is a simple extension of the sum of two partitions.

Definition 14. The sum of two PDCs, $\psi = \pi_1 + \pi_2$, on a set S is a PDC such that

- 1) s_i and s_j are in the same non-distinguished block of ψ if and only if there exists a sequence in S , $s_i = s_0, s_1, \dots, s_n = s_j$ for which, for $0 \leq l \leq n-1$, s_l and s_{l+1} are in the same non-distinguished block of either π_1 or π_2 and
- 2) s_i and s_j are in the distinguished block of ψ if and only if they are in the distinguished blocks of both π_1 and π_2 .

Example: $\pi_1 = (1,2)(3)(4)(5)(6)<7,8,9>$
 $\pi_2 = (1)(2)(3,4)(7)<5,6,8,9>$
 $\psi = (1,2)(3,4)(5)(6)(7)<8,9>$

This definition of the sum brings together in one block any states that are in the same blocks of either π_1 or π_2 , but it does not achieve the other requirement for preserving minimum table size, since every block of ψ does not contain a block of both π_1 and π_2 . In the example, (7) contains no block of π_1 , and both (5) and (6) contain no block of π_2 . By the method of computing ψ the \leq relation will never hold between ψ and π_1 if π_2 has blocks contained in $B_d(\pi_1)$. The \leq relation between PDCs holds if and only if there exists a pair of partitions in the sets generated by the PDCs between which the \leq relation holds. In order to share code bits between two inputs, the \leq relation must hold between the π s used for those inputs and ρ . Thus it is essential that if a sum is to be used in finding sharing opportunities, then the \leq relation must be made to hold. In order to achieve this the concepts of quotients and block size sets are now introduced.

Definition 15. The relation weakly less than or equal holds between two PDCs π_1 and π_2 ($\pi_1 \leq_w \pi_2$) if and only if for every $B_i(\pi_1)$, $i \neq d$, there exists a $B_j(\pi_2)$ such that $B_i(\pi_1) \subseteq B_j(\pi_2) \cup B_d(\pi_2)$.

The weakly less than or equal relation does not require any relationship between the blocks entirely contained in the other PDC's distinguished block. As Theorem 3 has proven, this relation does not guarantee the existence of generated partitions between which the less than or equal relation holds.

Theorem 9. If π_1 and π_2 are PDCs on S and $\psi = \pi_1 + \pi_2$ then $\pi_1 \leq_\omega \psi$ and $\pi_2 \leq_\omega \psi$.

Definition 16. If π and ψ are PDCs on S and $\pi \leq_\omega \psi$ then ψ defines a quotient PDC ψ/π on the blocks of π and the members of the distinguished block of π such that if α and β are either $B_i(\pi)$ or $\{s_i | s_i \in B_d(\pi)\}$ then α and β are in the same block of ψ/π if and only if α and β are contained in the same block of ψ .

Example: $\pi_3 = (1,2,3)(4,5)(6)(7)(8)<9,10,11>$
 $\pi_4 = (1)(3)(4,6)(9)<2,5,7,8,10,11>$
 $\psi = (1,2,3)(4,5,6)(7)(8)(9)<10,11>$
 $\psi/\pi_3 = ((1,2,3))((4,5)(6))((7))((8))(<9>)<10,11>$
 $\psi/\pi_4 = ((1)(3)<2>)((4,6)<5>)(<7>)<8>((9))<10,11>$

The notation used simply breaks up each block of ψ into its component blocks and don't-care states of π_3 or π_4 . Thus, the block $((1)(3)<2>)$ of ψ/π_4 shows that the non-distinguished blocks (1) and (3) plus the member $<2>$ of the distinguished block of π_2 are contained in block (1,2,3) of ψ .

Definition 19. If π and ψ are PDCs such that $\pi \leq_\omega \psi$ then the block size set of the quotient PDC, $BSS(\psi/\pi)$, is an array of integers a_i such that a_i is the number of non-distinguished blocks in $B_i(\psi/\pi)$.

Example: (continued)

$$BSS(\psi/\pi_3) = 1,2,1,1,0$$

$$BSS(\psi/\pi_4) = 2,1,0,0,1$$

Theorem 10 If $\psi = \pi_1 + \pi_2$, $\psi \leq \psi'$ and $B_d(\psi') \subseteq B_d(\psi)$, then $\pi_1 \leq \psi'$ if and only if $BSS(\psi'/\pi_1)$ includes no zeros. (Similarly for π_2).

Proof: There are no non-distinguished blocks of π_1 contained in $B_d(\psi)$ by the method of construction of ψ . Since $B_d(\psi') \subseteq B_d(\psi)$ there are no non-distinguished blocks of π_1 contained in $B_d(\psi')$.

If $\pi_1 \leq \psi'$ then, by the definition of \leq , since no non-distinguished blocks of π_1 are contained in $B_d(\psi')$, no non-distinguished blocks of ψ' are contained in $B_d(\pi_1)$. Therefore, for each $B_i(\psi')$ there is at least one $B_j(\pi_1)$ such that $B_j(\pi_1) \subseteq B_j(\psi') \cup B_d(\psi')$. By the method of constructing ψ and the fact that $\psi \leq \psi'$ it follows that $B_j(\pi_1) \subseteq B_i(\psi')$ and thus the i th element of $BSS(\psi'/\pi_1)$ is at least 1. Thus no element of $BSS(\psi'/\pi_1)$ is zero.

If $BSS(\psi'/\pi_1)$ contains no zeros, no non-distinguished block of ψ' is contained in $B_d(\pi_1)$. The construction of $\psi = \pi_1 + \pi_2$ assures that for each $B_i(\pi_1)$ there exists a j such that $B_i(\pi_1) \subseteq B_j(\psi)$. Since $\psi \leq \psi'$ and $B_d(\psi') \subseteq B_d(\psi)$ there exists a $B_j(\psi')$ such that $B_i(\pi_1) \subseteq B_j(\psi')$. Therefore, $\pi_1 \leq \psi'$.

Theorem 10 provides a method of constructing PDCs $\{\psi'\}$ from a sum of two PDCs π_1 and π_2 such that π_1 and π_2 are less than or equal to the PDCs, allowing the potential sharing of partitions generated from $\{\psi'\}$, between partitions generated from π_1 and π_2 . Since $\pi_1 + \pi_2 = \psi$, $\psi \leq \psi'$, and $B_d(\psi') \subseteq B_d(\psi)$, we must construct ψ' by joining blocks of ψ to form blocks of ψ' such that no block of ψ' has a zero in the block size set of the quotient of either π_1 or π_2 . This joining can be done by considering only the block size sets of π_1 and π_2 , treating their elements as pairs, and adding together

elements of the block size sets in order to eliminate zeros from the resulting block size sets. For example, it can be seen from $BSS(\psi/\pi_3)$ and $BSS(\psi/\pi_4)$ that if the fourth and fifth elements are joined and the first and third elements are joined, the resulting BSSs have no zeros. Thus $BSS(\psi'/\pi_3) = 2,2,1$ and $BSS(\psi'/\pi_4) = 2,1,1$. By joining the corresponding blocks of ψ , the result $\psi' = (1,2,3,7) (4,5,6) (8,9) <10,11>$ is realized. Figure 9 summarizes all of the upper bounds on such ψ' for π_3 and π_4 . Every other ψ' with similar properties can be shown to be greater than one of those listed, while the less than or equal relation does not hold between any pairs of ψ' given.

By means of sums of pairs of input PDCs, all those PDCs which can be shared by each pair can be found. An additional constraint must be applied to these PDCs to guarantee that the second condition of minimum bit, minimum table size definition is not impossible to satisfy. This condition is that the number of bits needed to encode the substring used for each input must be equal to the smallest number of bits required to encode that input partition. Consider the π_3, π_4 example as an application of this constraint. Since π_3 has 5 blocks it needs three bits to encode.

Similarly π_4 needs 2 bits. The number of bits needed in the shared encoding is the log of the number of non-distinguished blocks in the shared PDC plus the log of the maximum block size set element in the quotient. Figure 10 summarizes these statistics for the π 's found for π_3 and π_4 .

Note that the fact that one of the lower bounds doesn't satisfy condition 2 does not preclude a larger PDC from satisfying it. The BSS provides enough information for deciding. For example, the first ψ' , $(1,2,3) (4,5,6) (7,8,9) <10,11>$ does not satisfy condition 2. The BSSs are 1,2,2 and 2,1,1. The number of bits used for π_3 and π_4 by this PDC are 3 and 3 respectively. π_4 must use only 2. The quotients each use 1 bit. Producing a larger PDC can only join blocks. Thus the quotients remain the same or larger. However, if the number of blocks in ψ is reduced to 2, the number of bits in the quotient may become 2 for π_3 and remain 1 for π_4 . Thus the problem becomes one of combining elements of the BSSs in such a way that the $BSS(\psi/\pi_3)$ elements do not go above 4 and the $BSS(\psi/\pi_4)$ do not go above 2. There is only one way in which this can be done, by combining the second and third blocks, giving $\psi' = (1,2,3) (4,5,6,7,8,9) <10,11>$, and $BSS(\psi'/\pi_3) = 1,4$, $BSS(\psi'/\pi_4) = 2,2$.

The following definition formalizes the constraints on PDCs shared between two or more input PDCs. The extension from two to more than two is a simple generalization. PDCs which are themselves sums of input PDCs may be added to give PDCs which may be shared between all components of the sum, provided a suitable encoding string can be found.

Definition 18. Let $\psi = \sum_{i=1}^n \pi_i$. Then ψ is compatible with π_i for $1 \leq i \leq n$ if and only if there exists a ψ' such that, $\psi \leq \psi'$, $B_d(\psi') \subseteq B_d(\psi)$, and, for all i , $\lceil \log_2(\#(\psi')) \rceil + \lceil \log_2(\max(BSS(\psi'/\pi_i)) \rceil = \lceil \log_2 \#(\pi_i) \rceil$ and no element of $BSS(\psi'/\pi_i) = 0$. Any such ψ' is said to be completely compatible.

The definition of compatibility leads, in the general case, to an enumerative search for a ψ' with the properties listed. This search, while

difficult in general, may be eliminated if the existence or impossibility of such a ψ may be established by the application of several theorems.

Consider $\psi = \pi_1 + \pi_2$ and let S_{12} be the set of blocks of ψ with both non-zero BSS elements, S_1 the set of blocks with zero elements in BSS (ψ/π_2), and S_2 those with zero elements in BSS (ψ/π_1). The largest number of blocks a ψ' which satisfies compatibility may have is $\#(S_{12}) + \min(\#(S_1), \#(S_2))$ since no elements of the BSSs may be zero. It is an easy task to find if a ψ' with that many blocks exists. Let $c = \lceil \log_2(\#(S_{12}) + \min(\#(S_1), \#(S_2))) \rceil$ be the number of bits used in common between π_1 and π_2 with such a ψ' . Then $r_1 = \lceil \log_2(\#(\pi_1)) \rceil - c$ is the number of bits which remain to encode π_1 , while $r_2 = \lceil \log_2(\#(\pi_2)) \rceil - c$ is the number of bits which remain to encode π_2 . Assume without loss of generality that $\#(S_1) \leq \#(S_2)$. Then the blocks of ψ with non-zero elements in BSS(ψ/π_1) are not combined at all, and blocks of S_1 are each combined with a block of S_2 . Only blocks of S_2 in excess of $\#(S_1)$ remain to be combined with other blocks. These blocks all have elements in BSS(ψ/π_1) of zero, and therefore do not effect compatibility with π_1 . In addition, they have elements of 1 in BSS(ψ/π_2), and therefore are arbitrarily distributable in any blocks of ψ' with room for more π_2 blocks. The amount of room in each block B_i is simply 2^{r_2} - the i th element of BSS(ψ/π_2). If the total amount of room over all the blocks of S_{12} and S_1 for blocks of π_2 is larger than the number of excess blocks of S_2 , then ψ is compatible with π_1 and π_2 .

Theorem 11. Let $\psi = \pi_1 + \pi_2$. Assume $\#(S_1) \leq \#(S_2)$. ψ is compatible with π_1 and π_2 if every element of BSS(ψ/π_1) $\leq 2^{r_1}$, every element of BSS(ψ/π_2) $\leq 2^{r_2}$, and $(\#(S_{12}) + \#(S_1)) \cdot 2^{r_2} \geq \sum BSS(\psi/\pi_2)$.

Proof: From above discussion.

Theorem 12. Let $\psi = \sum_{i=1}^n \pi_i$. ψ is not compatible with π_i , $1 \leq i \leq n$ if there exists an i such that $\lceil \log_2(\#(\pi_i)) \rceil = \lceil \log_2(\max(\text{BSS}(\psi/\pi_i))) \rceil$.

Proof: If the number of bits available for π_i is equal to the number of remaining bits other than the common bits to encode π_i then there must not be any common bits. Hence π_i is not compatible with ψ .

Other theorems may be developed to improve the search for a compatible ψ . However, these are only necessary for automating these techniques in a computer based optimization program. These will be described in a subsequent paper. Examples to be described in this paper may be easily done by hand without large searches.

In the discussion of analysis techniques it was seen that a ρ_j of a subset encoding is greater than or equal to the sum of τ_i s for which it is used, while each τ_i is the product of the ρ_j s encoding it. The sum has been generalized to a sum of PDCs such that a PDC ψ_j may be part of an encoding if and only if it bears a specified relationship to the sum of the π_i s for which it is used. A similar generalization of product is needed to judge whether a subset of ψ_j s may be used to encode a specific π_i . The following product operation is defined to provide a necessary condition for the set of ψ_j s to ultimately yield ρ_j s whose product τ_i is in $G(\pi_i)$. The distinguished blocks of the ψ s plus possible combining of blocks may subsequently destroy this possibility.

Definition 19. The product of two PDCs $(\psi_1 \cdot \psi_2)$ is the PDC ψ such that

- 1) s_i and s_j are in the same non-distinguished block of ψ if they are in the same non-distinguished block of both ψ_1 and ψ_2 , and 2) s_i is in $B_d(\psi)$ if s_i is in $B_d(\psi_1)$ or s_i is in $B_d(\psi_2)$.

Theorem 13. In a partially specified subset encoding $\psi_1 \psi_2 \dots \psi_k$ of input PDCs π_1 through π_n , such that each ψ_j is completely compatible with all $\pi_i \in C_j$, a necessary but not sufficient condition for the existence of a minimum table size subset encoding $\rho_1 \rho_2 \dots \rho_k$, such that $\rho_i \in G(\psi_i)$ for all i , with identical row and column sets R and C , is that for all π_i ,

$$\pi_i \leq_s (\prod_{\psi_j \in R_i} \psi_j).$$

Proof: None of the non-distinguished blocks of π_i are contained in the distinguished block of $\prod \psi_j$. This arises from the fact that B_d of each ψ_j is contained in B_d of the sum of all π_k in C_j , which is the intersection of the $B_d(\pi_k)$ s. Therefore no non-distinguished block is contained in any $B_d(\psi_j)$ for $\psi_j \in R_i$.

The requirement for a minimum table size subset encoding is that $\pi_i \leq_s (\prod_{\rho_j \in R_i} \rho_j)$ for all i . Since all the blocks of π_i already appear in the blocks of $\prod_{\psi_j \in R_i} \psi_j$ the assignment of states in $B_d(\pi_i)$ in the various ψ_j s cannot cause the blocks of π_i to either be split apart or joined together in the product. Thus $\pi_i \leq_s (\prod_{\psi_j \in R_i} \psi_j)$ is a necessary condition for $\pi_i \leq_s (\prod_{\rho_j \in R_i} \rho_j)$ if $\rho_j \in G(\psi_j)$.

This condition is not a sufficient condition because some don't-care states of the ψ_j s may be already assigned in such a way as to cause them to form a block or blocks of their own in the final product of ρ_j s, while they are in the don't-care block of the product of ψ_j s.

The following example illustrates the insufficiency of Theorem 13.
Let $\pi_i = (1)(2)(3)(4)(5) \langle 6,7 \rangle$,

$$R_i = \{(1,2,3,7) \underline{(5,6)} (4), (1,3,5)(2,4,6,7), (1,2,5) (3,4) \langle 6,7 \rangle\}$$

$$(\prod_{\psi_j \in R_i} \psi_j) = (1) (2) (3) (4) (5) \langle 6,7 \rangle = \pi_i.$$

However, note that the product of the first two ψ_j s = (1,3)(2,7)(4)(6). No matter where the 6 is assigned in the third ψ_j , it will not bring 6 into a block with another block of π_i . Therefore no $(\prod_{\rho_j \in R_i} \rho_j)$ exists in which $\rho_j \in G(\psi_j)$.

Note that the 7 is not a problem, since it may be assigned with the (1,2,5) block of the third ψ_j and still remain within the block (2,7) of the product, which contains the (2) block of π_i .

V. PAIRWISE COMPATIBILITY AND POSSIBLE PROFILES

Compatibility concepts are now used to form a strategy for finding an optimal encoding. In the process of determining compatibility, a search is made for the largest c such that the number of blocks, 2^c , may contain the elements of the BSSs without violating any of the constraints. If $c > 1$, the PDCs considered are compatible. If this is done for all pairs of input PDCs, then the maximum pairwise overlap of input PDCs is found. In addition, new upper and lower bounds on the total number of bits may be obtained.

Theorem 14. Let c_{ij} be the maximum bit sharing between input PDCs π_i and π_j . The largest $t_{ij} = \lceil \log_2(\#(\pi_i)) \rceil + \lceil \log_2(\#(\pi_j)) \rceil - c_{ij}$ over all i and j where $i \neq j$ is a lower bound on the number of bits in the encoding. The smallest $\sum t_{ij} + \sum \#(\pi_k)$ such that every input appears at least once as an i, j , or k is an upper bound on the number of bits in the encoding.

Proof: t_{ij} is simply the total number of bits required to encode π_i and π_j if they are allowed to overlap to the largest degree possible. No encoding

may be smaller than the required number of bits necessary to encode any two of the input PDCs.

An upper bound is found by assuming that no further sharing of bits than pairwise can take place. Then we may choose pairs (and any remaining single input PDC) to cover all the input PDCs with the least total number of bits.

The strategy for choosing an encoding will be illustrated with the example state table given in Figure 2, with input PDCs given in Figure 7.

The following PDCs are the pairwise sums of the input PDCs of Figure 7 and their compatibility status. Details are shown in some cases only.

$$\psi_{1,2} = \pi_1 + \pi_2 = (1,3)(2,7)(4)(5)(6)$$

$$\psi_{1,2}/\pi_1 = ((1,3)) ((2,7)) (<4>) (<5>) ((6))$$

$$\text{BSS}(\psi_{1,2}/\pi_1) = 1,2,0,0,1$$

$$\psi_{1,2}/\pi_2 = (<1,3>) ((2,7)) ((4)) ((5)) (<6>)$$

$$\text{BSS}(\psi_{1,2}/\pi_2) = 0,1,1,1,0$$

$$\text{compatible: } r_1^{1,2} = 1, c_{1,2} = 1, r_2^{1,2} = 1, \text{ BSS becomes } \begin{matrix} 2,2 \\ 1,2 \end{matrix}$$

$$\psi_{1,3} = \pi_1 + \pi_3 = (1,3)(2)(4)(5)(6,7)$$

$$\text{compatible: } r_1^{1,3} = 1, c_{1,3} = 1, r_3^{1,3} = 1, \text{ BSS becomes } \begin{matrix} 2,2 \\ 2,2 \end{matrix}$$

$$\psi_{1,4} = \pi_1 + \pi_4 = (1,2,3,4)(6)(7)<5>$$

incompatible: Theorem 12.

$$\psi_{2,3} = \pi_2 + \pi_3 = (1,3)(2,6,7)(4)(5)$$

$$\text{compatible: } r_2^{2,3} = 1, c_{2,3} = 1, r_3^{2,3} = 1, \text{ BSS becomes } \begin{matrix} 1,2 \\ 2,2 \end{matrix}$$

$$\psi_{2,4} = \pi_2 + \pi_4 = (1,2,4,7)(3)(5)<6>$$

compatible: $r_{2,4}^{2,4} = 1, c_{2,4} = 1, r_{4,4}^{2,4} = 0$, Theorem 11. BSS becomes $\begin{matrix} 2,1 \\ 1,1 \end{matrix}$

$$\psi_{3,4} = \pi_3 + \pi_4 = (1,2,3,4)(5)(6,7)$$

incompatible: Theorem 12.

The largest t_{ij} of these is $t_{1,2} = 3$. Therefore, the smallest possible encoding is three bits. Since $t_{1,3} + t_{2,4} = 5$, five bits is an upper bound. The search for the optimal encoding begins with the $\psi_{i,j}$ of the lower bound, and overlaps the other PDCs on this in all possible ways, subject to the maximum overlaps computed in the above sums. Only when no encodings of the smallest size are shown to be possible are encodings of larger sizes considered. A possible profile is a binary matrix with one column for each bit in the encoding and one row for each PDC included in the encoding. Ones in the matrix indicate which bits are assigned to the PDC. The input PDCs are first ordered in descending $\lceil \log_2(\#(\pi_i)) \rceil$ order in order to insert the more difficult PDCs first. The first two PDCs are those giving the lower bound. The resulting order for this example is $\pi_1, \pi_2, \pi_3, \pi_4$.

The subset encoding to be illustrated is further constrained to be a substring encoding, which is a subset encoding in which each subset is contiguous. This constraint is used in order to produce encodings that can be easily implemented by a single shift operation followed by a single AND operation on many conventional computers, whether large, mini, or micro. It is not easy, in general, to assemble an arbitrary subset into a table index. All methods illustrated hold for a subset encoding as well, therefore the illustration does not lose generality.

The first partial possible profile is shown in Figure 11, and labeled pp1. PDCs are assigned from left to right. π_1 may be assigned anywhere in the three bits. π_2 must then satisfy the overlap constraints. Thus the leftmost position in which it can be assigned starts at bit 2, since it can only overlap π_1 in one bit. ($c_{1,2} = 1$).

When π_3 is added to the profile, it is found that it must either overlap π_1 or π_2 in two bits, which is impossible since both $c_{1,3}$ and $c_{2,3}$ are 1.

Since π_3 cannot be added to the partial profile, we must backtrack, to see if we can modify π_1 and π_2 to allow insertion of π_3 in three bits. Since π_2 cannot move to the right and moving π_1 to the right will only produce mirror images of previous profiles, π_3 cannot be added in three bits.

Four bit profiles are generated next. We obtain pp2 and pp3 by inserting π_1 , π_2 , and π_3 at their leftmost positions. Since π_4 is incompatible with π_1 and π_3 , and π_1 or π_3 are in every position, π_4 cannot be inserted into this partial profile. We therefore backtrack to pp2, move π_2 to the right, and insert π_3 in its leftmost position. We are then able to insert π_4 in the last bit, for pp6.

This profile includes all PDCs and satisfies the overlap conditions generated above. The overlap conditions tell us that if the two PDCs overlap, there exist partitions such that a minimum entry encoding can be achieved. However, we have no guarantee that the partitions used for the π_1 , π_3 overlap above are also useful for the π_2 , π_3 overlap, for example.

At this stage we must then test this "possible profile" to see if in fact we can find a minimum entry encoding (by our method of finding it, it is already minimum bit).

VI. EVALUATION OF POSSIBLE PROFILES

Each bit of the possible profile of an encoding must satisfy several constraints in order that it will eventually yield a partition in a minimum bit, minimum entry encoding. What is tested is whether the several constraints can be met simultaneously. Let us label the columns of the encodings as ψ_1, ψ_2, ψ_3 , and ψ_4 .

Each ψ_j must satisfy the constraint that it is completely compatible with all π_i in C_j . In addition, the constraints $\pi_i \leq_s (\pi_{\psi_j \in R_i})$ must hold for all π_i .

In the example under discussion, ψ_1 is only used for π_1 , and not shared with anything else. We therefore defer consideration until later.

ψ_2 must be a 2 block PDC, greater than $\psi_{1,3}$ since it is shared between π_1 and π_3 , and must have BSS elements 1 or 2. (None may be 0, and $r_1^{1,3} = r_3^{1,3} = 1$.)

$$\psi_{1,3} = (1,3)(2)(4)(5)(6,7) \quad ; \quad \text{BSS} = \begin{matrix} 1 & 1 & 0 & 0 & 2 \\ & & 1 & 0 & 1 & 1 & 1 \end{matrix}$$

Since no BSS element may be greater than 2, and there may be only two blocks in ψ_2 , the (1,3) and (2) blocks must be combined. All resulting π_3 BSS elements are 1, allowing the blocks (4) and (5) to be assigned to either of the other two blocks, but not together. Therefore,

$$\psi_2^1 = (1,2,3,4)(5,6,7) \quad \text{or} \quad \psi_2^2 = (1,2,3,5)(4,6,7).$$

In either case, BSS = 2 2.

2 2

Thus there are only two choices for ψ_2 . ψ_3 is examined next.

ψ_3 performs two functions. It must be greater than or equal to $\psi_{2,3}$.

since it is shared between π_2 and π_3 . In addition, together with ψ_2 it must give π_3 . In order to satisfy these constraints, we use the following information.

$$\psi_{2,3} = (13)(267)(4)(5) ;$$

$$\text{BSS} = 0 \ 1 \ 1 \ 1$$

$$1 \ 1 \ 1 \ 1$$

$$r_2^{2,3} = 1, c_{2,3} = 1, r_3^{2,3} = 1$$

From these numbers we can see that any two blocks may be brought together in one block of ψ_3 with the remaining two blocks brought together in the second.

We also know that

$$\psi_2^1 / \pi_3 = ((1,3)(4)<2>)((5)(6,7)) \text{ and}$$

$$\psi_2^2 / \pi_3 = ((13)(5)<2>)((4)(6,7)).$$

Each of these quotients tell which blocks of π_3 must be separated by ψ_3 in order that the product $\psi_2 \cdot \psi_3$ will give π_3 .

Let us arbitrarily choose ψ_2^1 . Then there are two choices for ψ_3

$$\psi_3^1 = (1,2,3,6,7)(4,5) \text{ or}$$

$$\psi_3^2 = (1,3,5)(2,4,6,7).$$

Each of these separate (13) from (4) and (5) from (6,7) and are greater than $\psi_{2,3}$ with BSS elements of 1 or 2.

Next, we examine $\psi_4 \cdot \psi_4$ must be greater than $\psi_{2,4}$.

$$\psi_{2,4} = (1,2,4,7)(3)(5)<6> ;$$

$$\text{BSS} = 2 \ 0 \ 1$$

$$1 \ 1 \ 0$$

Since all BSS elements must be 1 or 2, there is a unique compatible

$$\psi_4 = (1,2,4,7)(3,5)<6>.$$

The product of the two possible ψ_3 s and ψ_4 are now checked.

$$\pi_2 = (2,7)(4)(5) <1,3,6> \not\leq \psi_3^1 \cdot \psi_4 = (127)(3)(4)(5) <6>.$$

$$\pi_2 \not\leq \psi_3^2 \cdot \psi_4 = (1)(3,5)(2,4,7) <6>.$$

Failure of these products with respect to π_2 means that we must backtrack to any previous choice. Thus we must consider $\psi_2^2 = (1,2,3,5)(4,6,7)$. Using ψ_2^2 / π_3 and $\psi_{2,3}$ found above, we have two choices for ψ_3 ,

$$\psi_3^3 = (1,3,4)(2,5,6,7) \text{ or } \psi_3^4 = (1,3,2,6,7)(4,5).$$

Only one ψ_4 is available from the above computation. Thus we again compare the products $\psi_3^3 \cdot \psi_4$ and $\psi_3^4 \cdot \psi_4$ with π_2 , and again find

$$\pi_2 \not\leq \psi_3^3 \cdot \psi_4 \text{ and } \pi_2 \not\leq \psi_3^4 \cdot \psi_4$$

No other choices were made in the evaluation of this possible profile. Therefore there is no minimum table entry substring encoding with this profile.

The next step is to backtrack in the formation of a possible profile. None of π_4 , π_3 , or π_2 can move further to the right under the pairwise compatibility constraints, therefore π_1 must be moved. The possible profile pp7 in Figure 11 is obtained by placing π_2 , π_3 , and π_4 in the leftmost permissible position. This possible profile must be evaluated next.

First ψ_1 must be formed from $\psi_{2,4}$, which was observed above to have a single completely compatible PDC, $\psi_1 = (1,2,4,7)(3,5) <6>$. Since $\pi_4 \leq \psi_1$, no further product check needs to be done on the fourth row. $\psi_1 / \pi_2 = ((2,7)(4) <1>)((5) <3>)$. Therefore there are four choices for ψ_2 :

$$\psi_2^1 = (1,2,3,5,7)(4) \langle 6 \rangle$$

$$\psi_2^3 = (1,2,7)(3,4,5) \langle 6 \rangle$$

$$\psi_2^2 = (1,4)(2,3,5,7) \langle 6 \rangle$$

$$\psi_2^4 = (1,3,4,5)(2,7) \langle 6 \rangle.$$

ψ_2 is also constrained by $\psi_{1,2}$.

$$\psi_{1,2} = (1,3)(2,7)(4)(5)(6) \quad ; \quad \text{BSS} = 1,2,0,0,1$$

$$0,1,1,1,0$$

(1,3) and (2,7) must be in different blocks of ψ_2 . Therefore, only ψ_2^4 satisfies both the product and sum constraints, with $\langle 6 \rangle$ added to (1,3,4,5) to satisfy complete compatibility.

$$\psi_2^4 = \psi_2 = (1,3,4,5,6)(2,7).$$

ψ_3 is found next. $\psi_2/\pi_1 = ((1,3)(6) \langle 4,5 \rangle)((2)(7))$. Therefore ψ_3 must be either:

$$\psi_3^1 = (1,2,3)(6,7) \langle 4,5 \rangle \quad \text{or}$$

$$\psi_3^2 = (1,3,7)(2,6) \langle 4,5 \rangle.$$

ψ_3 must also be completely compatible with $\psi_{1,3}$.

$$\psi_{1,3} = (1,3)(2)(4)(5)(6,7) \quad ; \quad \text{BSS} = 1,1,0,0,2$$

$$1,0,1,1,1$$

From $\psi_{1,3}$, blocks (1,2,3) and (6,7) must be in different blocks of ψ_3 , leaving only ψ_3^1 . However, (4) and (5) must be assigned to make ψ_3 completely compatible with $\psi_{1,3}$.

The possibilities are:

$$\psi_3^3 = (1,2,3,4)(5,6,7) \text{ or}$$

$$\psi_3^4 = (1,2,3,5)(4,6,7).$$

Finally, ψ_4 need only satisfy the quotient of ψ_3/π_3 . There are a number of choices since

$$\psi_3^3/\pi_3 = ((1,3)(4)<2>)((5)(6,7)) \text{ and } \psi_3^4/\pi_3 = ((1,3)(5)<2>)((4)(6,7)).$$

Let us chose one possibility, ψ_3^3 and $\psi_4 = (1,2,3,5)(4,6,7)$.

Summarizing the ψ 's obtained:

$$\psi_1 = (1,2,4,7)(3,5)<5>$$

$$\psi_3 = (1,2,3,4)(5,6,7)$$

$$\psi_2 = (1,3,4,5,6)(2,7)$$

$$\psi_4 = (1,2,3,5)(4,6,7)$$

There still remains one don't-care in ψ_1 . This must be placed in such a way that it does not become a block of its own in any product involving ψ_1 . (Remember that a product check is a necessary but not sufficient condition for a solution. The only product in which ψ_1 is involved is $\pi_2 \leq_s \psi_1 \cdot \psi_2$. For a solution $\pi_2 \leq_s \rho_1 \cdot \rho_2$. $\rho_2 = \psi_2$ since there are no don't-cares in ψ_2 .

$\pi_2/\rho_2 = ((4)(5)<1,3,6>)((2,7))$. Therefore the $<6>$ must be placed either with (4) or (5) in ρ_1 to prevent a separate block (6) in the product. Since (4) or (5) are in both blocks of ψ_1 , $<6>$ can go into either block. Let us choose the first block.

Our final results are

$$\rho_1 = (1,2,4,6,7)(3,5)$$

$$\rho_2 = (1,3,4,5,6)(2,7)$$

$$\rho_3 = (1,2,3,4)(5,6,7)$$

$$\rho_4 = (1,2,3,5)(4,6,7).$$

To produce a state encoding and a set of tables, binary values must be assigned to the blocks of the ps. One arbitrary choice is

		p_1	p_2	p_3	p_4	
	1	0	0	0	0	} encoding
state	2	0	1	0	0	
	3	1	0	0	0	
	4	0	0	0	1	
	5	1	0	1	0	
	6	0	0	1	1	
	7	0	1	1	1	

Input 1	0	1	1	0	} profile
Input 2	1	1	0	0	
Input 3	0	0	1	1	
Input 4	1	0	0	0	

From the above and the original state table, the tables of Figure 4 are built in a straight forward manner.

REFERENCES

- [1] H. A. Sholl, "Direct Transition Memory and its application in computer design," IEEE Trans. Comput., C-23, pp. 1048-1061, Oct. 1974.
- [2] A. Grasselli and W. Montanari, "On the minimization of READ-ONLY memories in microprogrammed digital computers," IEEE Trans. Comput., (Short Notes), Vol. C-19, pp. 1111-1118, Nov. 1970.
- [3] A. J. Nichols, "An overview of microprocessor applications," Proc. IEEE, Vol. 74, pp. 951-953, June 1976.
- [4] C. Bass and D. Brown, "A perspective on microcomputer software," Proc. IEEE, Vol. 64, pp. 905-909, June 1976.
- [5] J. R. Millenson, "Language and list structure of a compiler for experimental control," Computer J., Vol. 13, pp. 340-343, Nov. 1970.
- [6] J. Hartmanis and R. E. Stearns, Algebraic Structure Theory of Sequential Machines, New York: Prentice-Hall, 1966.
- [7] P. Weiner and E. J. Smith, "Optimization of reduced dependencies for synchronous sequential machines," IEEE Trans. Elec. Computers, Vol. EC-16, pp. 835-847, Dec. 1967.

- [8] C. Harlow and C. L. Coates, "On the structure of realizations using flip-flop memory elements," Inform. and Control, Vol. 10, pp. 159-174, Feb. 1967
- [9] H. A. Curtis, "Systematic procedures for realizing synchronous sequential machines using flip-flop memory: Part I," IEEE Trans. Comput., Vol. C-18 pp. 1121-1127, Dec. 1969

Figure Captions

- Fig. 1. Conceptual state table representation.
- Fig. 2. Example state table.
- Fig. 3. State encoding and input substrings for example.
- Fig. 4. Realization for example.
- Fig. 5. Hardware implementation of interpreter.
- Fig. 6. PL/I implementation of interpreter.
- Fig. 7. Input PDCs, input table partitions, state encoding and input substrings using encoding algorithm I.
- Fig. 8. Input table partitions of Fig. 3.
- Fig. 9. Upper bounds of $\pi_3 + \pi_4$.
- Fig. 10. Compatibility statistics for the upper bounds of $\pi_3 + \pi_4$.
- Fig. 11. Possible Profile Search Trees.

TABLE LAYOUT OF FSM

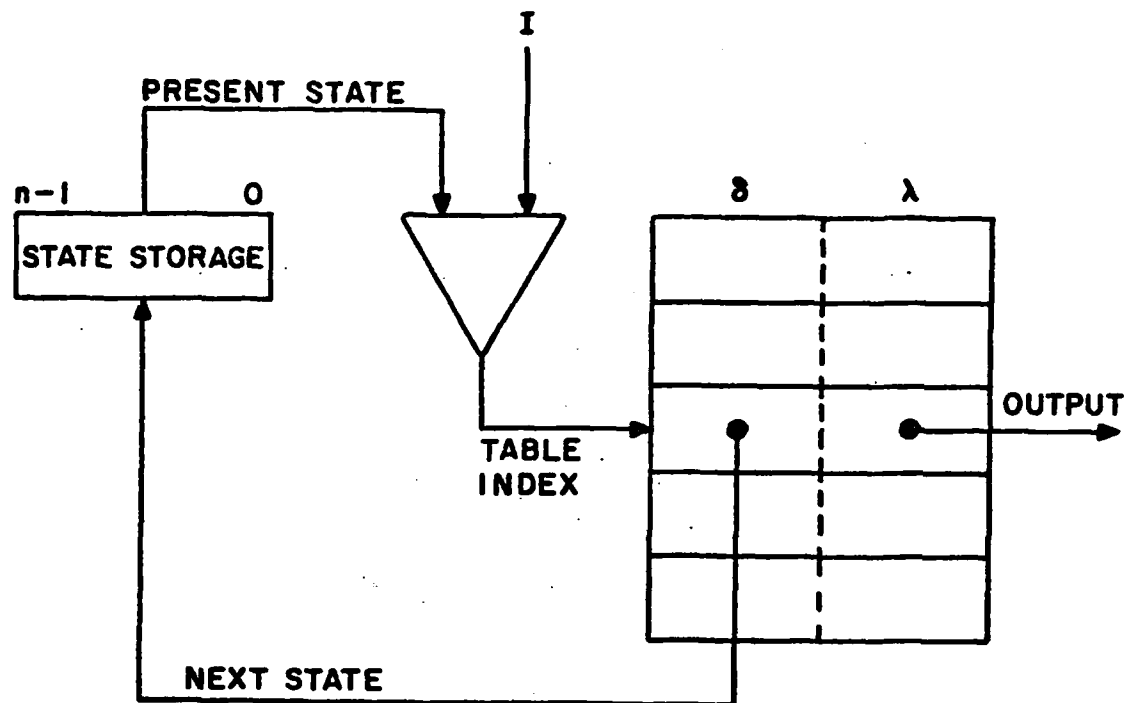


FIG. 1. CONCEPTUAL STATE TABLE REPRESENTATION.

STATE	INPUT			
	1	2	3	4
1	3A	-	4A	2A
2	1A	6A	-	2A
3	3A	-	4A	5A
4	-	5A	3B	2A
5	-	4A	4B	-
6	5A	-	3A	-
7	5B	6A	3A	-

FIG. 2. EXAMPLE STATE TABLE.

	y_1	y_2	y_3	y_4
1	0	0	0	0
2	0	1	0	0
3	1	0	0	0
4	0	0	0	1
5	1	0	1	0
6	0	0	1	1
7	0	1	1	1

STATE ENCODING

$I_1: y_2 y_3$

$I_2: y_1 y_2$

$I_3: y_3 y_4$

$I_4: y_1$

SUBSTRINGS

FIG. 3. STATE ENCODING AND INPUT SUBSTRINGS FOR EXAMPLE.

	NEXT STATE	OUTPUT	
TABLE +0	1000	A	I1 SUBTABLE
	1010	A	
	0000	A	
	1010	B	
+4	1010	A	I2 SUBTABLE
	0011	A	
	0001	A	
+7	0001	A	I3 SUBTABLE
	1000	B	
	0001	B	
	1000	A	
+11	0100	A	I4 SUBTABLE
	1010	A	

INPUT	1	TABLE
	2	TABLE +4
	3	TABLE +7
	4	TABLE +11

FIG. 4. REALIZATION FOR EXAMPLE.

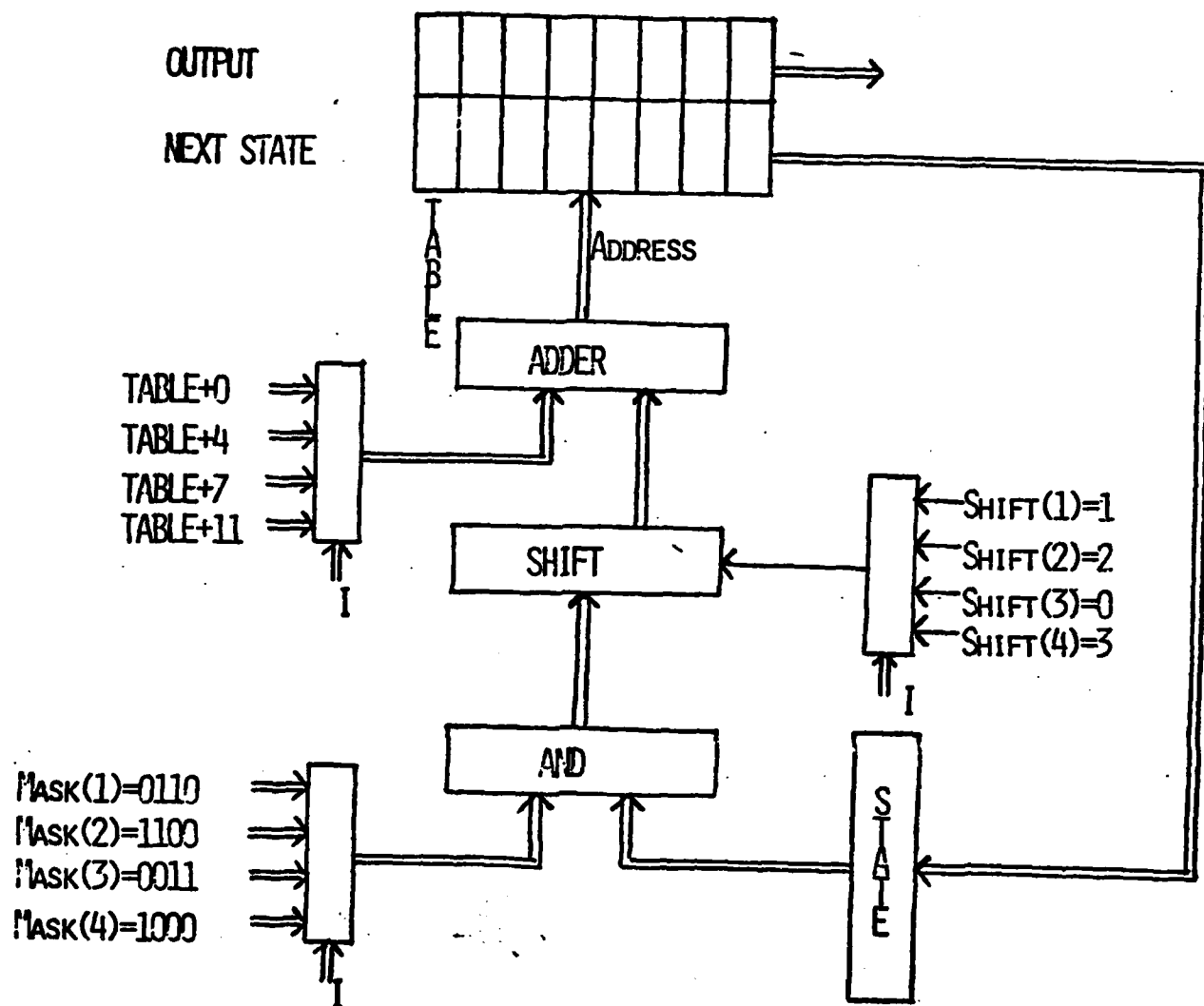


FIG. 5. HARDWARE IMPLEMENTATION OF INTERPRETER.

```

FSM: PROCEDURE (I);
  DCL FIRST_BIT (1:4) BINARY (16) INIT (2,1,3,1),
  LENGTH (1:4) BINARY (16) INIT (2,2,2,1),
  SUBTABLE (1:4) BINARY (16) INIT (0,4,7,11),
  NEXT_STATE_TABLE (0:12) BIT (4) INIT
    ('1000'B, '1010'B, '0000'B, '1010'B,
     '1010'B, '0011'B, '0001'B,
     '0001'B, '1000'B, '0001'B, '1000'B,
     '0100'B, '1010'B),
  OUTPUT_TABLE (0:12) LABEL INIT
    ( A,A,A,B,
      A,A,A,
      A,B,B,A,
      A,A ),
  STATE_BIT (4),
  INDEX BINARY (16),
  I BINARY (16);

INDEX = SUBTABLE (I) + SUBSTRING(STATE,FIRST_BIT(I), LENGTH(I));
STATE = NEXT_STATE_TABLE(INDEX);
GOTO OUTPUT_TABLE(INDEX);
A: /* output routine A */
  RETURN;
B: /* output routine B */
  RETURN;
END;

```

Fig. 6. PL/I IMPLEMENTATION OF INTERPRETER.

$$\begin{aligned}\pi_1 &= (1,3)(2)(6)(7)<4,5> \\ \pi_2 &= (2,7)(4)(5)<1,3,6> \\ \pi_3 &= (1,3)(4)(5)(6,7)<2> \\ \pi_4 &= (1,2,4)(3)<5,6,7>\end{aligned}$$

input PDCs

$$\begin{aligned}\tau_1 &= (1,3,4)(2,5)(6)(7) \\ \tau_2 &= (1,2,7)(3,4)(5,6) \\ \tau_3 &= (1,2,3)(4)(5)(6,7) \\ \tau_4 &= (1,2,4,5)(3,6,7)\end{aligned}$$

input table partitions

S	$E(\tau_1 \tau_2 \tau_3 \tau_4)$						
	y_1	y_2	y_3	y_4	y_5	y_6	y_7
1	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0
3	0	0	0	1	0	0	1
4	0	0	0	1	0	1	0
5	0	1	1	0	1	0	0
6	1	0	1	0	1	1	1
7	1	1	0	0	1	1	1

$$I_1: y_1 y_2 = E(\tau_1)$$

$$I_2: y_3 y_4 = E(\tau_2)$$

$$I_3: y_5 y_6 = E(\tau_3)$$

$$I_4: y_7 = E(\tau_4)$$

state encoding

FIG. 7. INPUT PDCs, INPUT TABLE PARTITIONS, STATE ENCODING
AND INPUT SUBSTRINGS USING ENCODING ALGORITHM I.

$$\begin{aligned}\tau_1 &= (1,3,4)(2)(5,6)(7) \\ \tau_2 &= (1,4,6)(2,7)(3,5) \\ \tau_3 &= (1,2,3)(4)(5)(6,7) \\ \tau_4 &= (1,2,3,5)(4,6,7)\end{aligned}$$

FIG. 8. INPUT TABLE PARTITIONS OF FIG. 3.

ψ'

BSSs

1	(1,2,3) (4,5,6)(7,8,9) <10,11>	1,2,2 2,1,1
2	(1,2,3,8) (4,5,6) (7,9) <10,11>	2,2,1 2,1,1
3	(1,2,3) (4,5,6,8) (7,9) <10,11>	1,3,1 2,1,1
4	(1,2,3,7) (4,5,6) (8,9) <10,11>	2,2,1 2,1,1
5	(1,2,3) (4,5,6,7) (8,9) <10,11>	1,3,1 2,1,1
6	(1,2,3,7,8) (4,5,6,9) <10,11>	3,2 2,2
7	(1,2,3,9) (4,5,6,7,8) <10,11>	1,4 3,1

FIG. 9. UPPER BOUNDS OF $\pi_3 + \pi_4$.

ψ'	π_i	(1) $A=\#(\psi')$	(2) $B=\max(BSS(\psi'/\pi))$	$\lceil \log_2 A \rceil + \lceil \log_2 B \rceil$	satisfies constraint 2
1	π_3	3	2	3	no
	π_4		2	3	
2	π_3	3	2	3	no
	π_4		2	3	
3	π_3	3	3	4	no
	π_4		2	3	
4	π_3	3	2	3	no
	π_4		2	3	
5	π_3	3	3	4	no
	π_4		2	3	
6	π_3	2	3	3	yes
	π_4		2	2	
7	π_3	2	4	3	no
	π_4		3	3	

FIG. 10. COMPATIBILITY STATISTICS FOR THE UPPER BOUNDS OF $\pi_3 + \pi_4$.

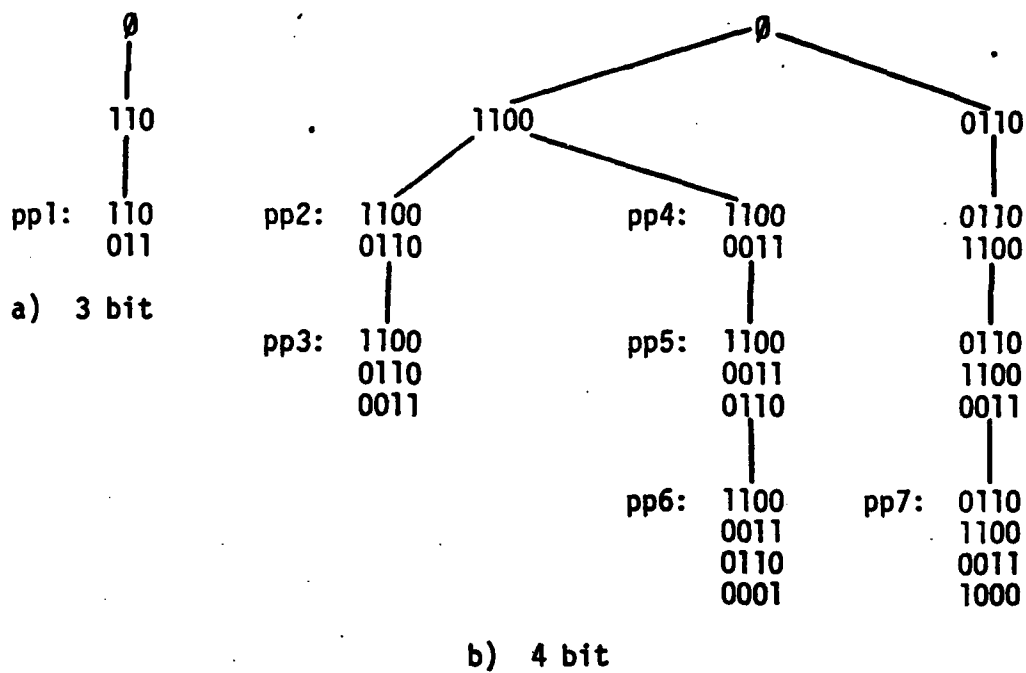


FIG. 11. POSSIBLE PROFILE SEARCH TREES.

FINITE STATE TECHNIQUES FOR SOFTWARE ENGINEERING SYSTEMS -
APPLICATIONS TO MICROCOMPUTER AND LARGE SCALE SYSTEMS

E.J. Wilkens

Department of Computer Science
Rutgers University
New Brunswick, New Jersey

TENTH HAWAII
INTERNATIONAL CONFERENCE
ON
SYSTEM SCIENCES
1977

FINITE STATE TECHNIQUES FOR SOFTWARE ENGINEERING SYSTEMS -
APPLICATIONS TO MICROCOMPUTER AND LARGE SCALE SYSTEMS

Edward J. Wilkens
Livingston College, Rutgers University
New Brunswick, New Jersey

Abstract

Finite-state techniques are applied to the specification of the control portion of a program. A language, an implementation representation, and an optimization are described.

1. INTRODUCTION

The problem of software design and engineering is with us at all levels of system size, from the microcomputer to the complex megacomputer system. One facet of software system design that seems to lend itself to a similar solution regardless of size is that of control. At the microcomputer level, control of a hardware system or peripheral is often the primary function of the microcomputer as a result of an ever decreasing crossover point in the cost of a microcomputer compared to a TTL controller [1]. At the other end of the size spectrum large systems often contain significant control programs or executives in addition to their host operating systems, as well as lower level I/O control routines [2,3]. This paper examines the application of finite state techniques in a set of programs designed to form the nucleus of a software engineering system to allow "high-level" specification of control programs and automatic conversion to efficient program.

2. FINITE STATE SPECIFICATION LANGUAGE

A Finite State Specification Language, FISSL, has been implemented. The primary objective of this

language is to enhance the user's ability to specify a finite state machine naturally and efficiently, and to understand its operation once specified. An intermediate representation of the machine is stored on disk to serve as input to the optimizer and other system programs.

The language is interactive, in which statements are interpreted line by line to modify the finite state machine as so far described. The language interpreter responds either with a diagnosis of error, or to obtain clarification of allowable ambiguities. For example, the use of a non-existent state would be an error, but the respecification of a table entry may or may not be intended by the user, so that clarification must be requested.

The main features of the language are as follows. Numbers in parentheses indicate line numbers in Figures 1, 3, and 5. There is a declaration statement to provide FISSL with disjoint sets of names for the inputs, states, and outputs of the finite state machine (1, 2, 3). The sets of names are kept disjoint in order to avoid ambiguity without demanding an ordering of names by type or use of type identifying keywords in the language state-

This work was supported by Grant #DAHCIS-73-G of the Advanced Research Projects Agency.

ments. The statements for the specification of the finite state machine's table entries are compact yet logically clear. The primary statement is

FOR <condition list> DO <table entry>

The condition list indicates in a very flexible way which entries in the table are to be specified by this statement. A single entry (4), an entire row or column (5), a set of entries produced by taking the cross product of a subset of states and a subset of inputs (6), or the union of several of these (11) may be easily specified. In addition an ALLBUT operator (9) allows easy specification of very large subsets. DC (5,7) indicates that an entry is to be left unspecified. The table entry part of the statement may be a state (10), an output (11), or a state and an output (6). If the state or output is omitted, it is left unspecified or with any previously specified value. Fig. 2 shows the state table specified by (1) to (13).

Another important feature of FISSL that allows compact specification is the ability to specify compound state names. A state may be divided into several substates (16), each of which has a set of names declared for it (17,18). The cross product of these sets of names is the machine's state set. Therefore, S1 becomes twelve states, named H.ST, H.TX, ..., T.ST, ..., TT.SYN2.

If one or more of the substate names is omitted in specifying the state in the condition list, then all possible values are used for the omitted substates in determining the table entries to be specified. For example, in (21), (TX,ST) specifies states H.TX, T.TX, TT.TX, H.ST, T.ST, and TT.ST. In the table entry part, TX indicates that the D substate becomes TX. If the P substate is unspecified in the table entry then no change in its specification is made. If SAME P is specified however, P is specified to remain the same in the next-state as in the present state. Thus in (21) H.ST has a next state of H.TX, while T.ST has a next-state of T.TX.

Fig. 3 specifies the state behavior of a Half Duplex Non-switched Multipoint Data Communication System originally presented by Bjorner [3]. Fig. 4

shows the resulting state table.

A quite significant extension is the addition of procedure-like statements. The term procedure-like is used because they do not specify a procedure that is used at run time, but at compilation time to build the state table. It is primarily built around a CASE statement. Fig. 5 gives a specification that is easily seen to be structurally identical to a decision process in a typical branching program.

Note that statement grouping is achieved by DO and OD, and all CASE statements have explicit ELSEs. The state may be modified anywhere, and output may be specified anywhere. Outputs become more complex if more than one output is encountered in reaching the lowest level of nesting. ERROR indicates that an error indication should be put in the state table and no further processing should be done on this path through the specification. Only the substates encountered with an explicit next state are specified. If a substate is not encountered in the table entry part, it remains unspecified, unless listed as a parameter of SAME. In (73) SAME REMAINING specifies that any substates not changed in the scope of the DO in (46) is to remain the same as it is specified in the condition-list. For example, the entry at I1, A2.B2.C3.D1.E2 receives explicit next state specification of A1.B3.C2. D and E remain to be specified, therefore they become D1.E2.

The state table is built by the compiler by repeated execution (simulation) of the specification.

The STATE set is the only one broken into substates here. Similar methods may be used for the INPUT set. Note however that the OUTPUT set has been implicitly changed by allowing more than one output to appear on a path through the nested CASE statements. For example, state A4.B2.C3.D2.E2 has an output of 04,05,02. Thus the actual outputs are ordered subsets of the specified OUTPUT set.

3. REPRESENTATION OF A FINITE-STATE MACHINE

This section assumes that a finite-state machine

has been produced and has been reduced so that no states are equivalent. The task is now to code this machine into a format capable of being loaded into a random access or read-only memory. An efficient storage consuming coding for the machine is desired. A table representation will be used since tables may be stored in random access memories efficiently and because multi-way branching can be easily achieved. Fig. 7 shows a conceptual interpreter and table implementation of a finite state machine. An input causes the present state to be read from state storage. A mapping function on the input and present state form an index into the table containing the next-state and output functions. The next-state is read from the table and written into state storage, while the output is used appropriately, usually as an address or routine to be called, for a large system, or as a binary word to be placed on a peripheral bus in a micro-computer system.

The method considered in this paper decomposes the finite-state machine into a sub-table per input, with an additional table of pointers to these sub-tables. The decomposition of the table into input sub-tables gives an opportunity for memory savings. It is assumed that the next-state and output entries are either both specified or both don't care.

If the input is known, then there is redundancy in the information provided by the state. For example, it is never necessary to identify a state whose entry is don't care, nor is it necessary to distinguish between two states with the same entries. Providing memory locations for this redundant information is wasteful. A state encoding scheme that would allow identification of a unique entry in the sub-table is desirable. That is, the original finite-state machine should be transformed to a table with only one occurrence of each distinct entry in each input sub-table.

Fig. 2 is a finite-state machine with four inputs, 1 through 4, seven states, A through G, and two outputs, X and Y. The input sub-table corresponding to 1 must contain four entries, one for each AX, CX, EX and EY. Similarly, the 4 sub-table must

contain 2 entries. The state number no longer provides a direct index to the sub-tables. For example, if the machine is in state G and input 1 is received, there is no 7th entry in the 1 sub-table. A mapping algorithm from the state to the input sub-table indices is needed. The method adopted here for its simplicity of use is to encode the states using binary variables $y_1 y_2 \dots y_n$ such that for each input there exists a substring of these binary variables which can be used as an index into the corresponding input sub-table. Fig. 8 shows an appropriate encoding of the states of Fig. 2. Since the 1 sub-table has four entries, the 1 substring should have four distinct codes, which are 00, 01, 10, and 11 under variables $y_2 y_3$. Note that the 2 substring $y_1 y_2$ has exactly 3 values. In addition, values of $y_1 y_2$ for any two states are the same if and only if the entries of those two states are the same or one or both is don't-care under 2 in Fig. 2. Thus states A and D have the $y_1 y_2$ value 00. Under input 2 state A is don't-care and state D is EX. Fig. 9 shows the tables which ultimately realize the machine of Fig. 2.

The interpreter for the substring encoding is either software or a circuit which masks the appropriate bits for each input, shifts them to the right, and uses them as an index into the subtable. By using such a circuit or software, there is little custom designed circuitry, none in the case of the software interpreter, and a circuit amenable to possible LSI integration with only the final mask and shift quantities to be determined for the individual state table, perhaps to be loaded into a Read Only Memory. This implementation has the important property that the input sub-table lookup constitutes a multiway branch to the output routine, which gives good real-time characteristics, and is in fact a major reason for the representation chosen.

4. STATE TABLE OPTIMIZATION PROGRAM

A State Table Optimization Program (STOP) has been written which compacts a state table into a set of minimal input sub-tables. The optimization depends on the finding of an appropriate encoding having a substring for each subtable. Such an en-

coding always exists but it may have more than the minimum number of bits necessary to uniquely encode the state set. The encoding used above uses 4 bits rather than the 3 bits necessary to uniquely specify 7 states. STOP is written to find the minimum bit encoding for minimum entry input sub-tables.

- (1) INPUTS 1,2,3,4
- (2) STATES A,B,C,D,E,F,G
- (3) OUTPUTS X,Y
- (4) FOR 2,G DO F,X
- (5) FOR 1 DO C,A,C,DC,DC,E,E
- (6) FOR (2,3),(A,C,E) DO D,X
- (7) FOR 2,(A,C) DO DC
- (8) FOR 2,0 DO F,X
- (9) FOR 1,(ALL BUT D,E,G):3,(F,G) DO X
- (10) FOR (D,F,G):3 DO C
- (11) FOR 3,(D,E):1,G DO Y
- (12) FOR (A,B,D):4 DO E,X
- (13) FOR 2,0:4,C DO E,X

FIG. 1. EXAMPLE SPECIFICATION

- (14) INPUTS SOH,A,SYN,OLE,STX,ITB,ETB,ETX
- (15) STATES START,TSTART,S1,ENDB,ENDX
- (16) PART S1=P,D
- (17) SUBSTATES P=H,T,TT
- (18) SUBSTATES D=ST,TX,SYN2
- (19) FOR START,SOH DO H,ST
- (20) FOR H,ST,SYN DO H,ST
- (21) FOR (TX,ST),A DO TX,SAME P
- (22) FOR TX,SYN DO SYN1,SAME P
- (23) FOR (SYN1,SYN2),SYN DO SYN2,SAME P
- (24) FOR (SYN1,SYN2),A DO TX,SAME P
- (25) FOR (TX,SYN1,SYN2),ETB DO ENDB
- (26) FOR (T,TX,T,SYN1,T,SYN2,TT,SYN2),ETX DO ENDX
- (27) FOR (N,TX,H,SYN1,H,SYN2,START),STX DO T,ST
- (28) FOR (TT,SYN1,TT,SYN2),A DO DC
- (29) FOR (TT,ST,TT,TX),(SOH,SYN,STX,ITB,ETB,ETX) DO TT,TX
- (30) FOR (N,SYN2,T,SYN2,START),OLE DO TSTART
- (31) FOR TSTART,STX DO TT,ST
- (32) FOR TT,ST,OLE DO TT,SYN1
- (33) FOR (TT,SYN1,TT,SYN2),OLE DO TT,TX
- (34) FOR TT,TX,OLE DO TT,SYN2
- (35) FOR TT,SYN1,SYN DO TT,ST

FIG. 3. HALF DUPLEX NON-SWITCHED MULTIPPOINT DATA COMMUNICATION SYSTEM-SPECIFICATION OF STATE BEHAVIOR

START	SOH	A	SYN	OLE	STX	ITB	ETB	ETX
H,ST	-	-	-	TSTART	T,ST	-	-	-
H,ST	-	H,TX	H,SYN1	-	T,ST	-	-	-
H,TX	-	H,TX	H,SYN2	-	T,ST	-	-	-
H,SYN1	-	H,TX	H,SYN2	TSTART	T,ST	-	-	-
H,SYN2	-	-	-	-	-	-	-	-
T,ST	-	-	-	-	-	-	-	-
T,TX	-	T,TX	T,SYN1	-	-	-	-	-
T,SYN1	-	T,TX	T,SYN2	-	-	-	-	-
T,SYN2	-	T,TX	T,SYN2	TSTART	-	-	-	-
TSTART	-	-	-	-	-	-	-	-
TT,ST	TT,TX	TT,TX	TT,SYN1	TT,SYN2	TT,SYN2	TT,SYN2	TT,SYN2	TT,SYN2
TT,TX	TT,TX	TT,TX	TT,SYN1	TT,SYN2	TT,SYN2	TT,SYN2	TT,SYN2	TT,SYN2
TT,SYN1	-	-	-	-	-	-	-	-
TT,SYN2	-	-	-	-	-	-	-	-
ENDB	-	-	-	-	-	-	-	-
ENDX	-	-	-	-	-	-	-	-

FIG. 4. STATE TABLE SPECIFIED BY FIG. 3.

	1	2	3	4
A	CE	—	—	—
B	AE	FX	—	—
C	CE	—	—	—
D	—	—	—	—
E	—	—	—	—
F	EA	—	—	—
G	FX	EA	—	—

FIG. 2. STATE TABLE SPECIFIED BY FIG. 1

- (36) INPUTS 11,12
- (37) STATES S1
- (38) PART S1 = A,B,C,D,E
- (39) SUBSTATES A = A1,A2,A3
- (40) SUBSTATES B = B1,B2
- (41) SUBSTATES C = C1,C2,C3
- (42) SUBSTATES D = D1,D2
- (43) SUBSTATES E = E1,E2
- (44) OUTPUTS 01,02,03,04,05,06,07
- (45) CASE INPUT FOR
- (46) 11 DO
- (47) CASE A FOR
- (48) A1 ERROR
- (49) ELSE DO
- (50) CASE A,B FOR
- (51) A2,B2 DO A1,C2,01 00
- (52) A4,B1 DO
- (53) 04
- (54) CASE C FOR
- (55) C1 DO A1,C2,D1,07 00
- (56) C2 DO A1,C3,D2,06 00
- (57) C3 DO A2,05 00
- (58) ELSE ERROR
- (59) DO
- (60) ELSE DO 00
- (61) B3
- (62) CASE D FOR
- (63) D1 DO 00
- (64) D2 DO
- (65) 01
- (66) CASE E FOR
- (67) E1 03
- (68) ELSE DO 00
- (69) DO
- (70) ELSE DO 00
- (71) 02
- (72) 00
- (73) SAME REMAINING
- (74) DO
- (75) 12 etc.

FIG. 5. SPECIFICATION USING CASE

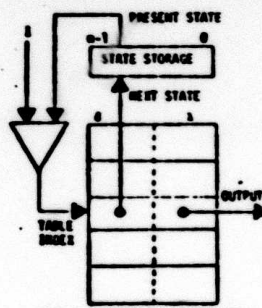


FIG. 7. CONCEPTUAL STATE TABLE REPRESENTATION.

	V ₁	V ₂	V ₃	V ₄
A	0	0	0	0
B	0	1	0	0
C	1	0	0	1
D	0	0	1	0
E	1	1	0	0
F	0	0	1	1
G	0	1	1	1

STATE ENCODING

FIG. 8. STATE ENCODING AND INPUT SUBSTRINGS FOR FIG. 3.

INPUT	11
STATE	ABCD
11111	ERROR
11112	ERROR
...	...
21322	ERROR
22111	13211/12
22112	13212/12
22121	13211/132
22122	13212/12
22211	13211/12
22212	13211/12
22221	13211/132
22222	13212/12
23111	13211/12
23212	13212/12
23221	13211/132
23222	13212/12
31111	ERROR
...	...
32322	ERROR
41111	13211/472
41112	13212/472
41121	13211/4732
41122	13212/472
42111	13311/4632
42112	13312/462
42211	13311/4632
42222	13312/462
43111	23311/452
43112	23312/452
43211	23311/4532
43222	23312/452
42111	ERROR
...	...
42322	ERROR

FIG. 6. PARTIAL STATE TABLE SPECIFIED BY FIG. 5.

TABLE	00	01	02	03	04	05	06	07	08	09	10	11
1	1000	1010	0000	1010	1010	0001	0001	0001	1000	0001	1000	0100
2	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1

FIG. 9. REALIZATION FOR FIG. 3.

FINITE STATE TECHNIQUES IN SOFTWARE ENGINEERING

E.J. Wilkens

Department of Computer Science
Rutgers University
New Brunswick, New Jersey

FINITE STATE TECHNIQUES IN SOFTWARE ENGINEERING

Edward J. Wilkens

Interdata, Inc.
106 Apple St.
Tinton Falls, N.J. 07724

Abstract

This paper examines the use of finite state techniques in a general, rather than application specific, context. It discusses applicability, benefits, and trade-offs, as well as techniques for embedding them in an overall system. It presents a specification language and a table optimization program which are tools for the use of these techniques.

Introduction

The problem of software design and engineering is with us at all levels of system size, from the microcomputer to the complex megacomputer system. One facet of software system design that seems to lend itself to a similar solution regardless of size is that of control. At the microcomputer level, control of a hardware system or peripheral is often the primary function of the microcomputer as a result of the ever decreasing crossover point in the cost of a microcomputer compared to a TTL controller [1]. At the other end of the size spectrum, large systems often contain significant control programs or executives in addition to their host operating systems, as well as lower level I/O control routines [2,3]. This paper examines the application of finite state techniques in several programs designed to form part of a software engineering system to allow high level specification of control functions and automatic conversion to efficient programs.

Many papers have appeared in the literature in the past decade describing the use of a finite state machine as an integral part of a computer program. Among the authors and applications were Beistand [2], an executive system with a complex control and decision structure, Johnson et. al. [4], lexical analysis, Millenson [5], the control of psychological experiments, and Birke [3] and

=====

This research was supported by the Defense Advanced Research Projects Agency under grant DAMRIS-73-G6 to the Rutgers project on Secure Systems and Automatic Programming. The author was formerly with the Department of Computer Science, Livingston College, Rutgers University.

Bjorner [6], teleprocessing device control and protocols. Each of these papers describes a specific use of a finite state machine, and explains to some extent the characteristics that make a finite state machine suitable.

A primary motivation in these papers for the use of finite state machines is their convenience for describing complex processes. Each was then transformed into the implementation that the author had determined was the most efficient according to the application's objectives. In most cases the specification of the finite state machine was an exact and reasonably understandable one.

It is the purpose of this paper to consider the use of finite state machines from a more general point of view. In taking this point of view, the paper concentrates on the benefits, trade-offs, and practical considerations involved in order to make the usefulness of the finite state machine in a given application more generally answerable than simply matching the application against the examples previously presented in the literature.

The paper presents a high level, compact, efficient language for the specification of a finite state machine, as well as an optimization program for a broad class of implementations.

A recent paper by Salter [7] presents another general approach to the use of finite state machines as part of a decomposition methodology. This paper takes a similar approach to Salter, but describes it from a different point of view, including more specific language and implementation detail.

Applicability of Finite State Machines

Many stored program systems are programmed to respond to an input from a single source or from one of a class of sources in the following manner. A stored indication of the past history of the input source and the present input are used to determine some action to be taken and an appropriate updating of the historical record to be made. In many cases the historical record has fixed size, and does not grow with the length of the input string, (number of inputs received over

time) leading to an informal characterization of such a system as a finite state machine.

Such systems are often implemented by assigning variables to various facets of the historical record, usually in a manner that is primarily understood by the programmer rather than efficiently utilized by the machine. The entire historical record, or state if we apply finite state machine terminology, is contained in many such variables. For a problem of any complexity, the program to respond to the receipt of an input does not decide on the action to be taken in a single N-ary decision, but usually branches through a sequence which examines one variable or relationship between variables at a time, making binary decisions along the way. On the other hand, an N state finite state machine may be considered to make a single N-ary decision on each input received.

If only a finite amount of historical record must be kept regardless of the length of the input sequence, then the historical record may be considered to be a state initialized to some initial value $S(0)$. At each input or event the next state (the new value of the state to be recorded) is a function of the current event and the present state, $NS(X,S)$. Similarly, the output at each event becomes a function of the current event and present state $O(X,S)$. These event oriented characteristics may be inherent in the system or they may be imposed by choice in a decomposition process. For example the application to lexical analysis [4] breaks up the input string into discrete characters, between which a state is stored. The decomposition of a problem into discrete sequential steps in this manner allows virtually any program to be cast as a finite state machine or a set of finite state machines.

Motivation for the Use of Finite State Techniques

The benefits derived from the use of finite state techniques fall into two classes, those that contribute to the ease of specification and testing of a system, and those that contribute to implementation efficiency.

Finite state techniques contribute to the ease of specification since they enable the description of the control of a system at a higher level than a direct programmed implementation of the control. They are at least one level of abstraction above a programmed specification. In addition, it is possible to carry over most techniques of the multiple levels of abstraction methodologies [8] into finite state techniques. This carryover is not described here, but is the subject of continued work.

The behavior of a finite state machine is easy to simulate prior to the availability of the detailed implementation of the machine itself and the subroutines which it calls. Therefore simulation testing of the system at a high level may begin as soon as the finite state machine(s) has been specified. Since the finite state

machine will be automatically transformed into its implementation, a high level of confidence in the implementation is possible after the simulation has been thoroughly tested. Of course, this confidence does not extend to the subroutines unless they too are specified in a manner automatically translatable into their implementation.

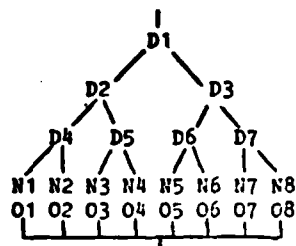


FIG. 1
REGULAR DECISION STRUCTURE

The primary implementation efficiency to be gained by these techniques is reduced decision time. The sequential decision process described above requires many binary decisions. Consider a very regular program graph, Figure 1, with k decisions in each path through the program. Assume that this program is run in response to one of the inputs to the system. Then this program is equivalent to one input column of a state table. Each node in the graph labeled with a D represents a decision, or branch, while updating of the state and output, marked with Ns and Os, are done after all decisions have been made. In Figure 1, k is

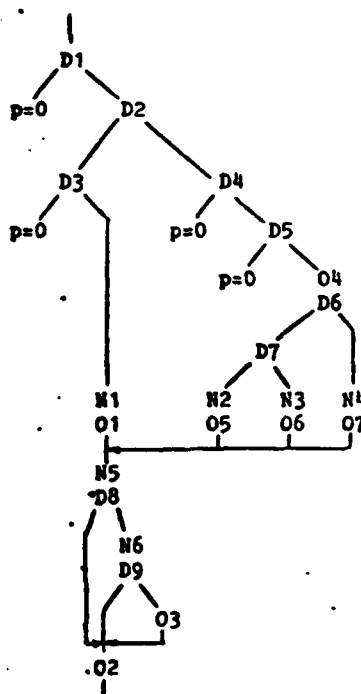


FIG. 2
TYPICAL DECISION STRUCTURE

4. If a constant time expenditure is assumed for both the decision of the program and the N-ary decision (state table lookup) of the finite state machine, then the finite state machine uses time a factor of k less than the binary decision structure. In a program not so regular in structure, there is some average number of decisions per entry into the program. In such a case the time savings is this average number. Figure 2 shows a typical graph, actually produced by the author as part of a commercial telephone system. In addition to points of branching, it also includes points of joining. Some of the exit points are marked $p=0$, meaning that this exit is provided as a defensive technique, and not normally used. If all other paths are equally likely, there is an average k of $6 \frac{2}{3}$.

Time may also be saved in recording the next state, since there will be only one updating of the state, using the value read from the state table. A multiple variable state will usually be updated at several different points in the program. The N's and O's in Figure 2 indicate points of updating the state and performing output, respectively. Thus the state is updated in $2 \frac{2}{3}$ places on the average entry into the program.

The remaining aspect of the entire process is the output. No time can be saved here, since the output must be performed regardless of the decision structure.

The final benefit obtained is the potential use of formal techniques in the coding and optimization process. Techniques are available from the current body of switching theory or may be newly developed for the special characteristics of program embedded finite state machines. These techniques are all decidable, although costly, which is not the case for general program schema.

As an example, existing techniques include state reduction [9]. The techniques which exist must be tempered by practicality. Most seek an optimal solution at the expense of computation time and in general may not handle machines as large as the ones which will result from the use of finite state techniques in programming. However, modifications which produce "good" reductions with moderate expenditure of computation time should be achievable.

A newly developed technique described below [10] finds an optimal memory consuming implementation which still preserves the N-ary branch property of a finite state machine.

Embedding a Finite State Machine in a System

Some previous work involving finite state machines in the programming process may be described as the replacement of specific subroutines at a relatively low level. Birke's device control programs [3], Bjorner's protocols [6], and Johnson's lexical processors [4] may be so described. On the other hand, in Heistand's

executive system [2] the finite state machine was the higher level decision maker, and the remaining system processes were called as subroutines by the finite state machine. The approach here is closer to the latter.

Finite state machines, with appropriate supporting programs and algorithms, form the highest level of decision makers. The state should include sufficient information for the decision made to be a significant part of the system's decision process. The finite state machine calls specific programs as output. Some of these output programs may be finite state machines as well, resulting in a hierarchical system. As an example, Heistand's executive could call Birke's device control. In addition, should the system complexity be such that a single finite state machine becomes too large, then several communicating finite state machines at the same level of hierarchy may be used. Such a decomposition often arises naturally along the usual system module divisions.

The classical definition of a finite state machine includes:

- a finite set of states, S ,
- a finite set of inputs, X ,
- a finite set of outputs, Z ,
- a next-state function, $NS(x,s)$, mapping X and S onto S , and
- an output function, $O(x,s)$, mapping X and S onto Z .

In order to allow the use of such a machine in a programming context, several additions must be made to this definition. The first addition arises from the fact that all information used by a program should not be considered state information. State information should be restricted to that needed for the next-state and output decisions to be made. Information needed by the system, but not for these functions, should not be allowed to expand the state set. This information is referred to as Task Data, and represented by the set D . An example of task data is the identification of a specific device. Without such information, no communication to the specific device could be performed. However, it is not needed to decide what actions to perform in response to an input from that device.

The addition of task data allows the removal of part of a system's function from the classical finite state machine definition for reasons of efficiency. Further removal may be achieved by allowing the finite state machine to use "algorithmic processes" where there are more efficient in specifying and performing the system's function. For example, consider a counting function, as in resource allocation. A large number of a resource may be available, and their number stored as part of the state. If this is done, a large number of states results, since each possible number of resources must be combined with every other facet of the state. Each time an input causes one of these resources to be allocated, the next-state is one that indicates

one less of the resource than the present state.

As an alternative, the number of the resource available may be recorded as task data. However, this number does play a part in the next-state and output decisions whenever it is zero, and must be changed each time one of the resource is allocated or returned. To make this possible, two sets of algorithms, input and output, designated IA and OA, are added to the finite-state machine definition. An input algorithm is used upon receiving an input, but before applying the next-state or output functions. The output algorithms are called by the output function. An input algorithm has access to the input and task data, and may modify the input, the task data or the state. The output algorithm has access to the state and task data, and may modify state, task data, or output.

The resource counting example may be handled with an output algorithm. Whenever a state change occurs that causes an allocation of one of the resource, an output algorithm is called which decrements the counter in task data, and tests it for zero. If it is zero, the algorithm must change the state to indicate none of the resource remains. Thus the possible states of the system include a substate with values of resource-present / resource-absent, but not with values of each possible number.

The fact that an input algorithm may change the input leads to the definition of an additional input set, the Modified Input, X' . The input to a program may have thousands of possible values, but only a small number of equivalence classes of these values cause distinct system reactions. These equivalence classes are then considered to be members of the modified input set, while the actual value of the input may be kept in task data if there is further need for it.

For example, an input may be a value from an analog to digital converter. The occurrence of an input from the converter is in the input set, while the value itself is task data received with the input. If there are three meaningful ranges of the value, say less than 10, from 10 to 100, and greater than 100, these three ranges are members of the modified input set. An input algorithm is called each time a value is received, which produces one of the modified inputs. The modified input is used for the next state and output functions.

A second example is that of an input which is a complex, multi-field record. X' should not include all possible values of this record. An appropriate and efficient input algorithm may be specified by a decision table, where the actions of the decision table are the values of modified input and task data to be used.

The definition of a finite state machine as extended here is summarized as follows:

a finite set of states, S ,

a finite set of inputs, X ,
a set of task data, D ,
a finite set of modified inputs, X' ,
a finite set of input algorithms, IA ,
mapping X and D onto X' , D and S ,
a next-state function, $NS(x', s)$, mapping
 X' and S onto S ,
a finite set of output algorithms, OA ,
mapping S , D , and Z
onto S , D , and Z .
a finite set of outputs, Z ,
an output function, $O(x', s)$, mapping
 X' and S onto Z and OA .

Note the difference in the use of decision tables and finite state machines above. Decision tables are used where the decision is to be made on the basis of new and unencoded data (for table lookup). Decision tables are converted into programs which avoid interpreting all fields of the record if possible to provide either memory or time efficiency. However, when the data upon which a decision is to be made is state data, already interpreted at earlier events, decision tables lose their usefulness, since an encoded state may be saved and used directly in a table lookup scheme.

The objective in formulating these changes in the definition of a finite state machine is to achieve a structure which allows sufficient flexibility to preserve practicality in diverse situations. With this definition, the classical finite state machine core may be completely overshadowed by the input and output algorithms. Obviously, this is not the intent. The input and output algorithms should be used only where necessary, and the next state and output functions should be used as the primary decision making tool.

Finite State Specification Language

A Finite State Specification Language, FISSL, has been implemented. The primary objective of this language is to enhance the user's ability to specify a finite state machine naturally and efficiently, and to understand its operation once specified. An intermediate representation of the machine is saved to serve as input to the optimizer and other system programs.

The language is interactive, in which statements are interpreted line by line to modify the finite state machine as so far described. The language interpreter responds either with a diagnosis of error, or to obtain clarification of allowable ambiguities. For example, the use of a non-existent state would be an error, but the respecification of a table entry may or may not be intended by the user, so clarification must be requested.

In the description to follow, numbers in parentheses indicate line numbers in Figures 3, 5, and 7. The line numbers are not part of FISSL, but are used for reference only. Disjoint sets of names are declared for the input, state and output sets (1), (2), and (3). The sets of names are

kept disjoint in order to avoid ambiguity without demanding an ordering of names by type or use of type identifying key words in the language statements. The declaration of the input and state sets cause an array to be set up of the proper dimensionality to hold the next state and output functions.

The statements for the specification are compact, yet logically clear. The primary statement is of the form

FOR <condition list> **DO** <table entry>

The condition list indicates which entries in the table are to be specified by this statement. The condition list 2,G in (4) specifies a single entry. Omission of an input or a state, as in (5), indicates that an entire row or column respectively is being specified. By enclosing a subset of inputs or states or both in parentheses it is indicated that the cross product of these subsets is to be specified. For example (6) shows the condition list (2,3),(A,C,E) which specifies entries 2,A, 2,C, 2,E, 3,A, 3,C, 3,E. The union of several of the above entry specifications may be achieved by separating them with a colon. For

- (1) INPUTS 1,2,3,4
- (2) STATES A,B,C,D,E,F,G
- (3) OUTPUTS X,Y
- (4) FOR 2,G DO F,X
- (5) FOR 1 DO C,A,C,DC,DC,E
- (6) FOR (2,3),(A,C,E) DO D,X
- (7) FOR 2,(A,C) DO DC
- (8) FOR 2,B DO F,X
- (9) FOR 1,(D,E,F,G):3,(F,G) DO X
- (10) FOR (D,E,F,G) DO C
- (11) FOR 3,(D,E):1,G DO Y
- (12) FOR (A,B,D),4 DO E,X
- (13) FOR 2,D:4,C DO E,X

FIG. 3. EXAMPLE SPECIFICATION

	1	2	3	4
A	EX	EX	EX	EX
B	EX	EX	EX	EX
C	EX	EX	EX	EX
D	EX	EX	EX	EX
E	EX	EX	EX	EX
F	EX	EX	EX	EX
G	EX	EX	EX	EX

FIG. 4. STATE TABLE SPECIFIED BY FIG. 3

- (14) INPUTS SOH,A,SYN,DLE,STX,ITB,ETB,ETX
- (15) STATES START,TSTART,S1,ENDB,ENDX
- (16) PART S1-P-D
- (17) SUBSTATES P-H,T,TT
- (18) SUBSTATES D-ST,TX,SYN1,SYN2
- (19) FOR START,SOH DO H,ST
- (20) FOR H,ST,SYN DO H,ST
- (21) FOR (TX,ST),A DO TX,SYN,P
- (22) FOR TX,SYN DO SYN1,SYN2,P
- (23) FOR (SYN1,SYN2),SYN DO SYN2,NAME,P
- (24) FOR (SYN1,SYN2),A DO TX,SYN,P
- (25) FOR (TX,SYN1,SYN2),ETB DO ENDB
- (26) FOR (TX,SYN1,SYN2,TT,SYN2),ETX DO ENDX
- (27) FOR (H,TX,H,SYN1,H,SYN2,START),STX DO T,ST
- (28) FOR (TT,SYN1,TT,SYN2),A DO DC
- (29) FOR (TT,ST,TT,TV),(SOH,SYN,STX,ITB,ETB,ETX) DO TT,TX
- (30) FOR (H,SYN2,T,SYN2,START),DLE DO TSTART
- (31) FOR TSTART,STX DO TT,ST
- (32) FOR TT,ST,DLE DO TT,SYN1
- (33) FOR (TT,SYN1,TT,SYN2),DLE DO TT,TX
- (34) FOR TT,TX,DLE DO TT,SYN2
- (35) FOR TT,SYN1,SYN DO TT,ST

FIG. 5. HALF DUPLEX NON-SWITCHED MULTIPoint DATA COMMUNICATION SYSTEM-SPECIFICATION OF STATE BEHAVIOR

example, in (11) 3,(D,E):1,G specifies the entries 3,D, 3,E, and 1,G.

The **ALLPUT** operator provides easy specification of large subsets by telling what is not in it. For example, in (9) 1,(**ALLPUT** D,E,G) specifies 1,A, 1,B, 1,C, and 1,F. (This example was for illustration rather than for any specification advantage.)

The table entry part of the **FOR** statement specifies the next state or the output or both to be filled in at the positions specified by the condition list. In (10) only the next state C is specified, in (9) the output X, and in (5) the pair F,X. An entire entry may be explicitly made don't care by using the keyword **DC** as in (7). Statement (7) is restoring 2,A and 2,C to don't care after they were given other values in (6). This is an example of where a request for clarification would be made.

If the condition list consisted of a single row or a single column only, then the table entry may be a list of the same length as the row or

SOH	A	SYN	DLE	STX	ITB	ETB	ETX
START	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST
H,ST	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST
H,TX	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST
H,SYN1	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST
H,SYN2	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST	H,ST
T,ST	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST
T,TX	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST
T,SYN1	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST
T,SYN2	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST
TSTART	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST	T,ST
TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST
TT,TX	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST
TT,SYN1	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST
TT,SYN2	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST	TT,ST
ENDB	ENDB	ENDB	ENDB	ENDB	ENDB	ENDB	ENDB
ENDX	ENDX	ENDX	ENDX	ENDX	ENDX	ENDX	ENDX

FIG. 6. STATE TABLE SPECIFIED BY FIG. 5.

- (36) INPUTS 11,12
- (37) STATES S1
- (38) PART S1 = A,B,C,D,E
- (39) SUBSTATES A = A1,A2,A3,A4
- (40) SUBSTATES B = B1,B2
- (41) SUBSTATES C = C1,C2,C3
- (42) SUBSTATES D = D1,D2
- (43) SUBSTATES E = E1,E2
- (44) OUTPUTS 01,02,03,04,05,06,07
- (45) CASE INPUT FOR
- (46) 11 DO
- (47) CASE A FOR
- (48) AT ERROR
- (49) ELSE DO
- (50) CASE A,B FOR
- (51) A2,B2 DO AT,C2,01 DO
- (52) A4,B1 DO
- (53) 04
- (54) CASE C FOR
- (55) C1 DO A1,C2,D1,07 DO
- (56) C2 DO A1,C3,D2,06 DO
- (57) C3 DO A2,05 DO
- (58) ELSE ERROR
- (59) 01
- (60) ELSE DO DO
- (61) 03
- (62) CASE D FOR
- (63) D1 DO 01
- (64) D2 DO
- (65) 01
- (66) CASE E FOR
- (67) E1 03
- (68) ELSE DO DO
- (69) 00
- (70) ELSE DO DO
- (71) 02
- (72) 00
- (73) SAME REMAINING
- (74) 00
- (75) 12 etc.

FIG. 7. SPECIFICATION USING CASE

INPUT	11
STATE	ABOVE
11111	ERROR
11112	ERROR
21322	ERROR
13211/12	13211/12
13212/12	13212/12
22112	13211/132
22121	13212/12
22122	13212/12
22211	13211/12
22212	13211/12
22221	13211/132
22222	13212/12
22311	13211/12
22312	13212/12
22321	13211/132
22322	13212/12
31111	ERROR
32322	ERROR
41111	13211/472
41112	13212/472
41121	13211/4732
41122	13212/472
41211	13311/4132
41212	13312/482
41221	13311/4132
41222	13312/482
41311	23311/452
41312	23312/452
41321	23311/4532
41322	23312/452
42111	ERROR
42322	ERROR

FIG. 8. PARTIAL STATE TABLE SPECIFIED BY FIG. 7.

column. This allows specification of a whole row or column consisting of arbitrary, rather than identical, entries in one statement. For example, (5) specifies the next states for all of input 1.

Any table entries not specified in any statements remain don't care. Figure 4 shows the complete finite state machine specified by Figure 3.

Compound States. A feature of FISSL that allows compact specification is the ability to specify compound state names. A state may be partitioned into several substates, each of which has a set of names declared for it. In Figure 5, the state set is initially declared to have five elements in (15). Then (16) partitions state S1 into P and D. Elements of compound names are connected by a dot. (17) and (18) declare the elements of P and D respectively. S1 is replaced in the state set by the cross product of P and D. Since none of the other states declared in (15) is partitioned, they remain as simple states in the state set.

A compound state name may be used in a FOR statement in the same manner as a simple state. Thus (20) illustrates the use of state H.ST in both the condition list and table entry. The usefulness of compound states is found in the ability to specify multiple states by leaving out one or more of the substates. This is interpreted in a condition list as the specified names with all possible names of the unspecified substates. In (21) TX means H.TX, T.TX, and TT.TX. ST is similarly combined with all elements of P.

When one or more substates is specified in the table entry part, only the values of those substates are recorded in the next state. The remaining substates retain any previously specified value. In some cases, the desired state change consists of changing only one of the substates, and leaving the other(s) unchanged from their present state value. Leaving such substates unspecified does not accomplish this. The keyword SAME followed by a list of substates causes those substates to have the same value in the next state as it has in the present state for each entry specified by the condition list. (21) uses SAME P in the table entry part. The condition list specifies all states with TX or ST as values of D. These states are to have a next state of TX in substate D. Thus H.ST,A has a next state of H.TX, while T.ST,A has T.TX.

Figure 6 shows the complete state table specified by Figure 5. This state table specifies part of the state behavior of a Half Duplex Non-switched Multipoint Data Communication System originally presented with a flow graph by Bjorner [6].

Case Statement. The final feature of FISSL is a procedure-like CASE statement. It is called procedure-like because it is not executed at run time, but at compile time to build the state table much in the manner of Parnas [11]. Figure 7 gives an example of the use of a case statement. It has

a structure quite close to that of Figure 2, except that the case statements may "branch" in more than two ways. The outputs O1, O2, etc. are the same as those shown in Figure 2.

CASE statements may be nested, and compound statements grouped by DO and OD. The form of the CASE statement is

```

CASE <condition> FOR
  <value1> DO <table entry> OD
  <value2> DO <table entry> OD
  .
  .
  <valueN> DO <table entry> OD
ELSE DO <table entry> OD

```

<table entry> may be either a table entry as described above or another CASE statement. <condition> has the same form as a simple condition in a FOR statement, but with a substate name rather than a substate element name, and INPUT rather than an input name. The keyword INPUT causes the values expected in the CASE statement to be input values. (45) shows a nested case statement that uses INPUT. Only the I1 table entry part is shown. Figure 7 therefore shows the specification of one column of the state table. (47) causes values of substate A to be used.

<value> is a value which <condition> may take on. Thus in (48), if A has the value A1 then an error routine should be executed. This routine would be provided by the user. If the value of the present state is not A1, then the table entry is specified by the CASE statement at (50). This condition expects values from A.B. Finally in (51) if the present state has the value A2.B2 then the next state has A1 specified for A and C2 specified for C. In addition output O1 is specified. That completes the evaluation of the CASE at (51). (61) is the next statement grouped in the (49) ELSE. Therefore, to the values of A and C is added the specification of B3 for B. Then the CASE statement of (62) uses values of P. To illustrate the complete operation of Figure 7, I1,A4.B1.C2.D2.E3 results in a next state of A1.B3.C3.D1.E3. The E3 is not explicitly specified, but is copied from the present state by SAME REMAINING in (73), which completes the next state in any positions not yet specified in an outer CASE statement. Note that the output has resulted in a string of outputs, namely O4,O6,O2. Thus the CASE statement gives the ability to specify ordered sequences of outputs as the result of one state change. Figure 8 shows the column of the state table built by repeated execution of the CASE statement.

FISSL as described here only allows compound states. Compound inputs are another useful feature that is not included here. As mentioned above, the CASE statement provides compound outputs.

State Table Optimization Program

A technique recently developed [10] is a method of encoding a finite-state machine to allow a minimum storage expenditure without sacrificing real time advantages. A State Table Optimization Program (STOP) has been written which compacts a state table into a set of minimal input subtables.

The actual choice of implementation characteristics to be optimized are not as important as the ability to produce an optimizer which will work on state tables of a more useful size than may be handled by most switching theoretic techniques. I believe that useful state tables will have a number of inputs X number of states product in the range 50 to 1000. Above 1000, the complexity of the function being realized calls for modularization, just as large programs must be modularized to keep complexity manageable.

The characteristics chosen for optimization are real-time, total number of table entries, and number of bits per table entry, in that order. The table look up speed of a two dimensional array is not to be sacrificed for memory. Any optimization technique to reduce memory usage should preserve the inherent speed of this technique by allowing at most a simple transformation of the state to a table index, so that the columns may be reduced in size.

The approach taken reduces each column of the state table to an input sub-table having one unique entry for each distinct specified entry in the original column. Note that don't-cares (denoted by -) are eliminated, as well as redundant occurrences of entries in a single column, such as CX in column 1. The input 1 column in Figure 3 has four distinct entries, as well as several don't-cares and a duplicate entry. The resulting input sub-table produced by this approach has exactly four entries. In the example of Figure 3, a 28 entry table is condensed to four sub-tables with a total of 13 entries.

The method used to map the state onto a sub-table index is to assign a substring of the state code to each input. These substrings should overlap as much as possible to keep down the total number of bits in the state encoding. The substring is used directly as the index to the input sub-table associated with that input. Such a mapping is easily performed on computers with logical and shift instructions.

The optimization depends on finding an encoding having a substring for each subtable. Such an encoding always exist, but it may have more than the minimum number of bits necessary to uniquely encode the state set. The minimum bit encoding which will yield 13 entry sub-tables for Figure 3 uses 4 bits rather than the 3 needed to uniquely specify 7 states.

STOP has been written to find the minimum bit encoding for minimum entry input sub-tables. The algorithm employed is a highly pruned,

backtracking search. It is based on an extension to the partition methods of Hartmanis and Stearns [12] which allows don't cares to be specified as part of the partition. This algorithm is described in detail in [10].

References

- [1] Nichols, A.J., An Overview of Microprocessor Applications, Proc. IEEE 74, (June 1976), 951-953.
- [2] Heistand, R.E., An Executive System Implemented as a Finite-state Automaton, Comm. ACM 7, (Nov. 1964), 669-677.
- [3] Birke, D.M., State-transition Programming Techniques and Their Use in Producing Teleprocessing Device-control Programs, IEEE Trans. Commun. 20, (June 1972), 569-575.
- [4] Johnson, W.L., Porter, J.H., Ackley, S.I., Ross, D.T., Automatic Generation of Efficient Lexical Processors Using Finite State Techniques, Comm. ACM 11, (Dec. 1968), 805-813.
- [5] Millenson, J.R., Language and List Structure of a Compiler for Experimental Control, Computer J. 13, (Nov. 1970), 340-343.
- [6] Bjorner, D., Finite-State Automation - Definition of Data Communication Line Control Procedures, Proc. FJCC 1970, 477-491.
- [7] Salter, K.G., A Methodology for Decomposing System Requirements into Data Processing Requirements, Proc. 2 Conf. Soft. Eng., (Oct. 1976), 91-101.
- [8] Liskov, B.H., and Zilles, S.N., Specification Techniques for Data Abstractions, IEEE Trans. Soft. Eng. 1, (March 1975), 7-18.
- [9] Paull, M.C., and Unger, S.H., Minimizing the Number of States in Incompletely Specified Sequential Switching Functions, IRE Trans. Elect. Comp. 8, (sept. 1959).
- [10] Wilkens, E.J., Realizations of Sequential Machines Using Random Access Memory, IEEE Trans. Comp., to be published.
- [11] Parnas, D.L., State Table Analysis of Programs in an Algo-like Language, Proc. ACM Nat. Meeting, (1966), 391-400.
- [12] Hartmanis, J., and Stearns, R.E., Algebraic Structure Theory of Sequential Machines, New York: Prentice-Hall, 1966.